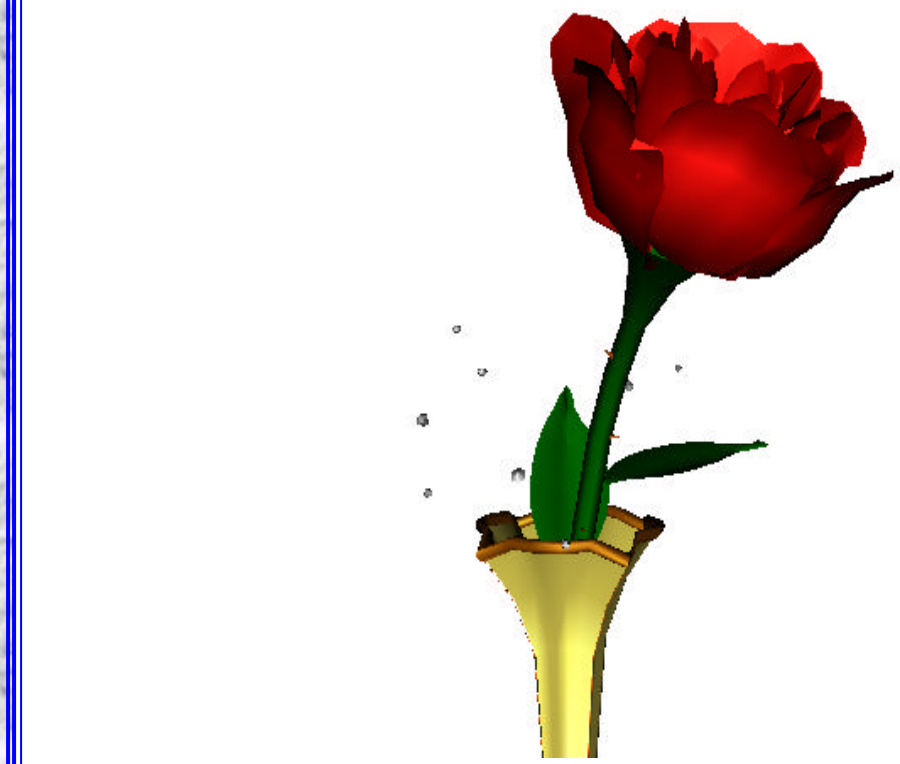
069

El Placer de Programar con VRML 2.0



Araujo Díaz David

México, D.F. año 2006

Contenido

Página

Prólogo

VRML son las siglas de **Virtual Reality Modeling Language** (lenguaje para el modelado de realidad virtual), es una de las herramientas más prometedoras en el campo de la visualización y graficación por computadora; no sólo porque es una herramienta gratuita, sino, por que además es un lenguaje sencillo e interactivo.

VRML, fue concebido como un lenguaje simple, que permitiera la creación de ambientes virtuales para **Internet** y en donde los usuarios pudieran interactuar con los objetos dentro de un **mundo virtual**, con eficiencia y buen desempeño. **VRML** permite la creación de gráficos impresionantes, animaciones, reproducción de sonido, enlaces con otros mundos y/o páginas de **Internet**, etc. Pero es posible aumentar su poder de cálculo al trabajar conjuntamente con otros lenguajes de programación como **Java** y **JavaScript**.

Las aplicaciones en las que es posible emplear **ambientes virtuales** son innumerables, desde las ciencias básicas, en entrenamiento, en capacitación, en visualización, en arquitectura, en ingeniería, en medicina, en biología, etc.

Este documento describe de forma breve algunas de las características más importantes de **VRML 2.0**, con las cuales es posible crear mundos interactivos, con sonido, animaciones, efectos de iluminación, etc. Además provee de una guía para la sintaxis de **VRML** y de la interacción de este con otros lenguajes como **JavaScript** y **Java**. Además presenta una breve introducción al lenguaje **X3D**, sucesor de **VRML**.

A continuación describimos el contenido de cada uno de los capítulos que componen este documento.

En el **capítulo 1** (de **Introducción**) se tiene una introducción histórica del surgimiento de **VRML** y se presentan las primitivas básicas, con las cuales es posible representar figuras geométricas, también se muestra como cambiar la apariencia (color) de los objetos y el fondo del universo.

Las **transformaciones** se describen en el **capítulo 2**, estas son importantes no sólo en **VRML**, sino en todos los mundos tridimensionales. Permiten modificar tres aspectos básicos de los objetos, como son su posición, rotación y escala. Este capítulo acaba con la descripción de los puntos de vista, necesarios para ir a puntos importantes dentro de los ambientes virtuales.

En el **capítulo 3** se muestran las diferentes formas en las que es posible representar **objetos**, como **puntos** en el espacio, mediante **líneas** (figuras de alambre), mediante facetas (objetos sólidos) y mediante el empleo de **mallas** para realizar objetos con formas complejas.

En el **capítulo 4** se presenta otro aspecto importante y divertido, la **iluminación**, **VRML** presenta tres tipos de iluminación, la ambiente (como la luz del Sol), la luz puntual (como la proveniente de un foco) y la luz Spot (como la de una lámpara). Aquí también se muestra la forma en la que es posible cambiar la textura de un objeto, colocándole una imagen a la figura seleccionada.

Un aspecto imprescindible en todos los lenguajes de programación en la posibilidad de definición de **prototipos**, es decir, objetos que es posible reutilizar y cambiar sus parámetros, este aspecto se discute en el **capítulo 5**, junto con los nodos `DEF` y `USE` que permiten repetir una parte del código que puede ser usado muchas veces.

Para lograr enlazar objetos a otros mundos o a páginas de **Internet**, se emplean los `anchors`, que se describen en el **capítulo 6**, junto con algunos efectos complejos como los `billboarding`, que permiten seguir la vista de un espectador dentro del mundo. También se muestran las **colisiones** y otros eventos.

El **capitulo 7** muestra dos herramientas interesantes, para la creación de animaciones, los **interpoladores** y los **sensores** de tiempo, proximidad, tacto, visibilidad y movimiento.

El **sonido** se describe en el **capítulo 8**; en **VRML** es posible agregar **sonido ambiental**, es decir, aquél que se escucha igual en todo el mundo y **sonido espacial**, que es aquél que se escucha de acuerdo a la posición relativa que presente el espectador respecto a la fuente sonora.

En el **capítulo 9**, se muestra la forma en la que es posible optimizar los ambientes virtuales a través del control del **nivel de detalle**.

Para aumentar el poder de **VRML** es posible enlazarlo con programas realizados en **JavaScript**, la forma de hacerlo se muestra en el **capítulo 10**.

No puede faltar la interacción con otros lenguajes más poderosos y de uso general, como **Java**, este aspecto se discute en el **capítulo 11**.

Para finalizar se presenta en el **capítulo 12**, una introducción a un lenguaje de graficación vectorial, **X3D**, que es el sucesor de **VRML**.

En el **apéndice A**, se resumen los aspectos básicos relacionados con las graficación, representación y transformación de objetos, en los espacios bidimensionales y tridimensionales, como una referencia rápida para realizar dichos cambios en los puntos que forman un objeto.

En el **apéndice B**, se tienen una serie de **preguntas de repaso**, cuyo objetivo es reafirmar y ampliar los conocimientos adquiridos a lo largo de la lectura de este documento.

Para los que se encuentren interesados en ampliar sus conocimientos de este lenguaje, se tienen al final las **referencias** usadas para la elaboración de este documento.

Espero que este documento sea de utilidad práctica y que permita a los principiantes conocer **VRML** y que a su vez sirva a las personas con experiencia en graficación, como guía de este lenguaje.

Todos los ejemplos, ejercicios y demás programas presentados fueron ejecutados y probados, y por lo tanto no deben de existir errores en su sintaxis, las imágenes presentadas también se capturaron directamente de la pantalla.

En la medida de lo posible se realizó una revisión detallada de cada uno de los ejemplos, ejercicios y programas, para que estos resultaran sencillos de entender, sin embargo, es posible aumentar rápidamente su complejidad, juntándolos.

También quisiera agradecer a mis compañeros y a los estudiantes que han tomado el curso, los comentarios y las sugerencias vertidas, realizadas a este documento, a los ejemplos y ejercicios.

Araujo Díaz David

México, D.F., año 2006

Capítulo 1: Introducción

Para la programación de ambientes virtuales existen diversas herramientas, entre las que se encuentra **VRML** (**V**irtual **R**eality **M**odeling **L**anguage) **[Alarcón 00]**. **VRML** permite la creación de mundos virtuales con objetos en tres dimensiones y el control de estos objetos a través de sensores de movimiento, tacto, visibilidad, proximidad y tiempo, permitiendo su manipulación y/o animación en el espacio virtual.

Los **objetos** realizados con **VRML** pueden ser visualizados desde un navegador de **Internet** (**Internet Explorer**, **Netscape Comunicator**, etc.), añadiendo visualizadores (**plug-in**) como **Cosmo Player**, **Platinum** ó el propio de **Microsoft**. Los archivos de programas **VRML** se pueden crear en cualquier **editor de texto**. Además, **VRML** puede interactuar con otros lenguajes de programación como **Java**, haciendo posible combinar una poderosa herramienta de visualización, con las ventajas de un lenguaje de programación de propósito general. **VRML**, fue desarrollado en el año de **1994** en **Silicon Graphics**, se tiene una revisión en **1997** con lo que surge **VRML 97** ó **VRML 2.0**.

VRML es un lenguaje de definición de escenas cuyo objetivo es la **descripción de entornos virtuales tridimensionales** (**3D**) que pueden transmitirse e interrelacionarse a través de **Internet**. La idea surgió en la primera **World Wide Web Conference**, en Ginebra, 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** (**Internet**).

Entre los asistentes se determinó la necesidad de disponer de un lenguaje simple capaz de describir una escena tridimensional, distribuíble a través de **HTTP** e integrable en el mundo **WWW** mediante la incorporación de hiperenlaces que permitieran saltar a otras escenas **3D** o a documentos **HTML**. Así, surgió el nombre de **VRML**, **V**irtual **R**eality **M**arkup **L**anguage, por analogía con el de **HTML** (**H**yper**T**ext **M**arkup **L**anguage). Posteriormente el nombre se cambio por el de **V**irtual **R**eality **M**odeling **L**anguage (**VRML**).

En la misma sesión se decidió comenzar el proceso de especificación del lenguaje y crear la lista de correo **www-vrml** con el objetivo de conseguir una primera especificación en unos cinco meses y así presentarla en la segunda conferencia **WWW** en octubre de ese mismo año. La lista, al final de su primera semana de existencia, ya contaba con más de mil suscriptores.

Tras una serie de debates en los que se discutieron diferentes propuestas, la alternativa presentada por **S**ilicon **G**raphics **I**nc. (**SGI**) fue la que consiguió un mayor número de votos. Esta propuesta consistía en utilizar como punto de partida el formato de archivos de **Open Inventor** (producto de dicha compañía), que era un conjunto de herramientas orientadas a objetos para el desarrollo de aplicaciones gráficas interactivas.

Gavin Bell (de **SGI**), tomo un subconjunto del formato mencionado, lo modificó de acuerdo a lo discutido en la lista de correo, y se le añadieron las extensiones para red. De esta forma, en octubre de **1994**, se presentó la primera versión de **VRML** que daría lugar a **VRML 1.0**.

A principios de **1996**, **Silicon Graphics** pone a disposición del dominio público **QvLib**, el cual se convirtió en el primer **interprete** de **VRML** (parser **VRML**); capaz de traducir el texto de una escena virtual a un formato entendible por un navegador. Posteriormente aparecería **WebSpace**, el primer navegador capaz de leer e interpretar todo el estándar **VRML**.

VRML 1.0 es un lenguaje para la descripción de mundos virtuales estáticos, que cumple tres requisitos fundamentales:

- Es **independiente** de la plataforma donde se ejecute el visualizador.
- Tiene capacidad para trabajar de un modo **eficiente** con conexiones lentas.
- **Extensibilidad**, es decir, facilidad para futuras ampliaciones del lenguaje.

El principal problema de esta versión de **VRML** se encontraba en la **ausencia de semántica**. Se trataba simplemente de un lenguaje de especificación de geometrías. Por tanto, a través de él, únicamente era posible recorrer mundos inertes. La ausencia de semántica llevaba incluso a situaciones inesperadas, como la posibilidad de atravesar los objetos representados, incluyendo suelos y paredes.

Debido a los problemas de la **versión 1.0**, enseguida surgió la **versión 1.1** del lenguaje que añadía algunas extensiones para la incorporación de sonido y animaciones simples. Paralelamente, la totalidad de las empresas que desarrollaban **navegadores VRML** incorporaban sus propias extensiones, comenzando por la detección de colisiones. Esto hizo peligrar la existencia de un estándar por lo que inmediatamente comenzó el trabajo para la especificación de la **versión 2.0** de **VRML**. Así, se creó el **VAG** (**V**RML **A**rchitecture **G**roup).

Seis propuestas fueron presentadas para la **versión 2.0** en las que se pretendía eliminar el problema de la semántica:

1. **Active VRML** de Microsoft.
2. **Dynamic Worlds** de **GMD** y otros.
3. **HoloWeb** de **SUN**.
4. **Moving Worlds** de **SGI** y otros.
5. **Out of this World** de **Apple**.
6. **Reactive Virtual Environment** de **IBM** Japón.

Tras el correspondiente proceso de estudio y revisión de las mismas se decidió aceptar la propuesta **Moving Worlds** de **SGI**, **Sony** y **Mitra**. La propuesta fue revisada en diversos borradores que dieron lugar a la versión definitiva de **VRML 2.0** en el otoño de **1996**. Esta versión aporta grandes mejoras sobre su predecesora, que se reflejan en las siguientes características:

- **Semántica.-** Esta es la principal mejora de la **versión 2.0**, que se consigue mediante la incorporación de nodos de comportamiento que pueden ejecutar scripts realizados en lenguajes como **C++**, **Java**, **JavaScript** y **VRMLScript**. Para ello es necesaria una comunicación entre nodos, comunicación que se realiza mediante eventos que se propagan a través de rutas (conexiones entre nodos generadores y receptores de eventos). De esta forma, es posible definir, por ejemplo, que al acercarse el usuario a determinado objeto, un nodo de proximidad genere un evento que sea recibido por un nodo de comportamiento, que ejecutará un programa adecuado.
- **Mayor poder descriptivo.-** La nueva **versión 2.0** de **VRML** permitía la definición de mundos más reales, al incorporar superficies irregulares, condiciones atmosféricas como niebla y nodos de sonido espacial.
- **Mayor interactividad.-** Mediante la incorporación de nodos sensores de proximidad, tacto, visibilidad y tiempo.
- **Animación.-** Mediante la incorporación de interpoladores.
- **Modularidad.-** Mediante la generación de prototipos que agrupan conjuntos de nodos predefinidos.

Finalmente, **VRML 2.0** se sometió al **ISO / IEC** (**I**nternational **O**rganization for **S**tandardization / **I**nternational **E**lectrotechnical **C**ommission), con el número **14772-1:1997** para su aprobación y estandarización. **VRML 97** es el resultado de ese proceso, y es una norma de **ISO / IEC**. **VRML 97** es prácticamente idéntico a **VRML 2.0**, con un par de cambios menores en detalles muy pequeños.

VRML 2.0 permite interactuar con el mundo virtual, sin embargo no es posible interactuar con otras personas que estén accediendo a ese mismo mundo en ese momento. El sucesor de **VRML** es **X3D** (E**x**tensible **3D**) que toma el trabajo seguido por el **VRML 97** y clarifica las zonas grises que no han sido cubiertas por la especificación a través de los años, así se brinda una mayor flexibilidad. Algunas modificaciones implican ser más precisos con la iluminación y los modelos de eventos, y cambiar el nombre de algunos campos para una mejor consistencia.

Los cambios importantes se pueden resumir en:

- Expansión de las capacidades gráficas.
- Un modelo de programación de aplicaciones revisado y unificado.
- Múltiple codificación de archivos para describir los mismos modelos abstractos incluyendo **XML** (E**x**tensible **M**arkup **L**anguage).
- Arquitectura modular que permite tener rangos de niveles de adopción y soporte para los distintos tipos de mercado.
- Expansión de la estructura de la especificación.

Una gráfica de la escena **X3D**, es idéntica a la gráfica de la escena **VRML 97**. El diseño original de la estructura gráfica de **VRML 97** y sus tipos de nodos ésta basada en la tecnología establecida, existente para gráficos interactivos. Esto sirvió a **VRML 97** bien por algunos años. Los cambios efectuados inicialmente en los gráficos **X3D** fueron para incorporar los avances del hardware comercial, a través de la introducción de nuevos nodos y nuevos tipos de campos para datos.

X3D proviene de la propuesta conocida como **Living Worlds** (Mundos Vivientes) de **Silicon Graphics**. En esta propuesta, todas las personas tienen un avatar (identidad virtual) que pueden definir utilizando las primitivas **VRML**. Este avatar es la representación del usuario en el entorno tridimensional, tanto para la detección de colisiones; como para el aspecto que el usuario tendrá para el resto de los visitantes. Se espera que se produzca una estandarización de avatares de tal forma que estos puedan ser empleados en distintos tipos de mundos.

1.1 Empleo de VRML en ambientes virtuales

VRML ha conseguido implantarse como herramienta de trabajo en numerosas disciplinas, a continuación mostramos una lista de algunas de ellas:

- Apoyo a Diseño de Automóviles.
- Apoyo a exploraciones planetarias.
- Arte Virtual.
- Manipulación de esculturas.
- Museos Virtuales.
- Bancos de Información Virtuales.
- Arquitectura Virtual.
- Edificaciones Virtuales.
    - Interiores Virtuales.
    - Usos de la Tierra.
    - Desarrollo Urbano.
    - Infraestructura de Servicios.
- Astronomía.
- Caminatas Virtuales.
- Control Virtual de Vuelos.
- Educación Virtual.
    - Arte.
    - Ciencia.
    - Geografía.
    - Historia.
    - Matemáticas.
    - Tecnología.
- Estudios Ambientales.
- Finanzas y Mercadeo.
- Historia Virtual.
    - Modelación de Estructuras y Pueblos Antiguos.
- Industrias Virtuales.
    - Modelos de Procesamiento.
    - Modelos de Transporte.
- Ingeniería Virtual.
- Juegos Virtuales.
- Medicina.
- Medios de Comunicación Masiva.
- Modelación de Reacciones Químicas y moléculas.
- Simulaciones de Sismos.
- Sitios de Reunión Virtuales.
- Telerrobótica.
- Visualización Científica.
- Visualización de Bases de Datos.

- Visualización de Ecosistemas.
- Visualización de Modelos Químicos.
- Visualización de Museos.
- Visualización de Redes.
- Visualización Educacional.
- Educación Virtual.

De todos los usos mencionados en la lista anterior podemos destacar **[Shneiderman 98]**:

- **Realidad Virtual en la medicina:** esto supone un importante avance en lo que a técnicas de formación se refiere. Médicos de todo el mundo podrán practicar delicadas operaciones en un mundo virtual diseñado a su medida pudiendo así perfeccionar las técnicas sin ningún riesgo. En concordancia con esto y con las nuevas tecnologías esta empezando a darse la **telemedicina**, que es la posibilidad de que un paciente sea atendido a distancia por su médico (**Figura 1.1**).

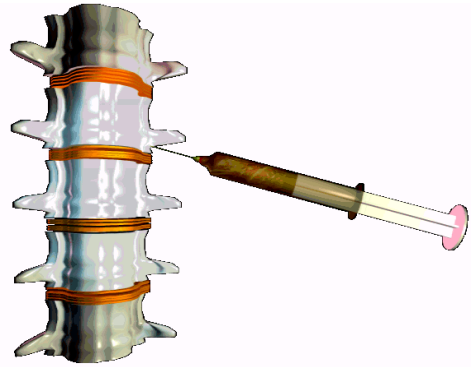


Figura 1.1: Realidad virtual en la medicina.

- **Realidad virtual en la arquitectura:** la arquitectura se desarrolla entorno al avance de la tecnología en lo que a mundos tridimensionales se refiere. Esto permite al arquitecto explorar virtualmente lo que va a ser su proyecto y mejorarlo si lo cree oportuno (**Figura 1.2**).

Figura 1.2: Realidad virtual en la arquitectura.

- **Realidad virtual en el arte:** poder contemplar las maravillosas obras artísticas presentes en todo el mundo esta fuera del alcance de muchos, cabe la posibilidad de contemplarlas en galerías virtuales diseñadas por ejemplo en **VRML**, que aunque no son lo mismo, siempre dejaran un mejor sabor de boca que contemplarlas sobre el papel y permitirán un estudio de las distintas obras un poco más detallado. Otro atractivo es la reconstrucción de obras que ya no existen o que se encuentran en ruinas, ó incluso crear obras de arte mediante la manipulación de imágenes virtuales (**Figura 1.3**).

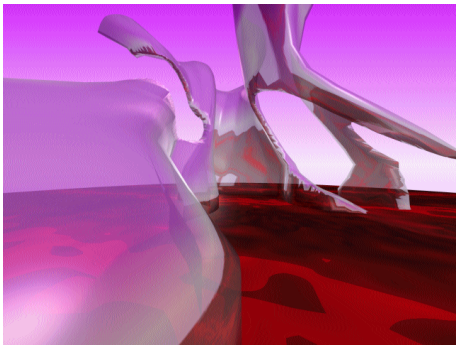


Figura 1.3: Arte Virtual.

- **Realidad virtual en animación:** este es un campo en el que el desarrollo no tiene límite y que va ligado a la imaginación del desarrollador. **VRML** sólo actúa como herramienta (**Figura 1.4**).

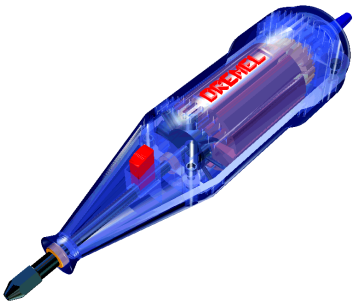


Figura 1.4: Realidad Virtual en la Animación.

- **Realidad virtual en la economía:** en los asuntos de decisión financiera, los ambientes virtuales ayudan a la consolidación de amplias cantidades de datos procedentes de diversas fuentes de referencia. La visualización tridimensional de este conjunto puede resultar muy valiosa para explicar el comportamiento de inversiones o transacciones a personas no familiarizadas con esa temática. Así mismo colabora con una efectiva manipulación y evaluación de conjuntos complejos de datos para efectos de aportar valiosos elementos de juicio a la toma de decisiones. **VRML** está siendo utilizado como medio para visualización financiera tridimensional porque su producto puede ser distribuido a través de la **Internet** y accedido por cualquier cliente.

- **Realidad virtual en la química:** es posible visualizar estructuras moleculares en **3D**. Esto supondría una importante herramienta para los químicos. Pero también es posible realizar y visualizar los procesos químicos, sin riesgo para las personas (**Figura 1.5**).

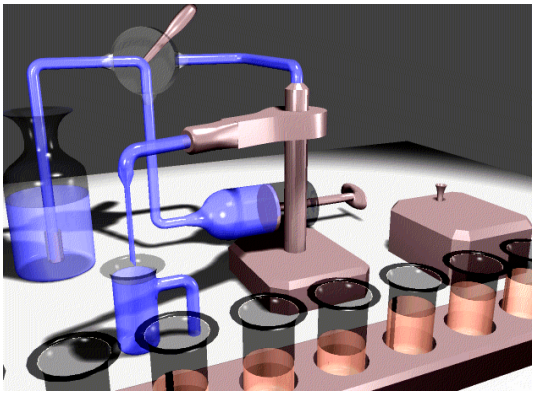


Figura 1.5: Realidad Virtual en los procesos químicos.

1.2 Creación de mundos VRML

Lo primero que debemos hacer para introducirnos en este apasionante mundo virtual es conseguir un **visor de VRML**. Si el sistema operativo es **Windows**, tendremos una gran oferta, aunque si usamos **UNIX/Linux** o **MacOS** tampoco tendremos ningún problema.

Actualmente existen dos opciones para realizar una navegación virtual:

1. La primera opción es a partir de un navegador especial independiente a los tradicionales como **Netscape** o **Internet Explorer**, que se ejecuta automáticamente cuando detectan la presencia de alguna aplicación **VRML**.
2. La segunda opción es la instalación de unos programas adicionales dentro de los navegadores. Estos programas son conocidos como **plug-in**, y se consideran la alternativa más utilizada por ser más fácil y cómodos de instalar. Si nuestra opción es ésta, visualizaremos nuestros mundos virtuales en el propio navegador y también nos permitirá algunas capacidades adicionales.

Criterios para la elección:

- **Precio.-** la mayoría de estos son gratuitos y se pueden descargar de **Internet**. Si nos exigen algún costo por ellos deben incluir alguna aplicación adicional.
- **Aceleración de hardware.-** hay algunos de los visores que aprovechan la aceleración de las tarjetas gráficas, algo que con el tiempo será mucho más aprovechado por los programadores de **VRML**.
- **Scripting.-** algunos visores también soportan tanto **Java** como **JavaScript**.

Los visores **VRML** originales eran programas independientes del navegador **WWW** tradicional, pero hoy en día la mayoría se incorporan a éstos en forma de **plug-in**. Existe multitud de visores para **VRML**, algunos de ellos se pueden encontrar en las siguientes direcciones (según en sistema operativo empleado):

Para **Windows**:

- Cosmo Player: **cic.nist.gov/vrml/cosmoplayer.html**
- GLView: **www.snafu.de/~hg**
- Live 3D: **wp.netscape.com/eng/live3d/windows/**
- Blaxxun: **developer.blaxxun.com/**
- BS Contact: **www.bitmanagement.com/**
- Flux Player: **www.mediamachines.com/products.html**
- FreeWRL: **freewrl.sourceforge.net/**
- Octaga Player: **www.octaga.com/**
- OpenWorlds: **www.openworlds.com/**
- Vcom3D: **vcom3d.com/Viewer.htm**
- Xj3D: **www.xj3d.org/**

Para **UNIX/Linux**:

- OpenVRML: **openvrml.sourceforge.net/**
- FreeWRL: **freewrl.sourceforge.net/**
- Octaga Player: **www.octaga.com/**
- Xj3D: **www.xj3d.org/**

Para **MacOS**:

- Cosmo Player: **cic.nist.gov/vrml/cosmoplayer.html**
- MacWeb3D: **www.macweb3d.org**
- FreeWRL: **freewrl.sourceforge.net/**
- Xj3D: **www.xj3d.org/**

Para crear mundos **VRML** se tienen dos opciones:

1. **Codificarlos a mano**, y para ello solo se necesita un editor del texto, como el **notepad** o el **wordpad**. Una vez realizado el código, se debe salvar como archivo `*.wrl`; pero debemos asegurarnos de guardar el archivo como **texto sin formato**, al igual que con los archivos `*.html`.
2. Se puede usar una de las muchas **herramientas de autoría** de **VRML** o **modeladores de 3D**. Esta opción es bastante interesante porque trabajar con **VRML** en crudo está muy bien, pero puede resultar difícil en algunas ocasiones al crear objetos complejos.

1.3 Formato de archivos

Los archivos de **VRML**, son archivos de **texto** (**ASCII**), por lo que pueden ser modificados en un editor de texto convencional [Alarcón 00]. Sin embargo es posible transformarlos en un **formato binario**, cuya principal ventaja es la ocultación del código fuente; otra ventaja es que su tamaño es más reducido.

También pueden estar **comprimidos** en formato `*.gz`, y ser enviados y ejecutados de esta forma.

Un archivo **VRML** tiene la extensión `*.wrl` (**world**).

La primera línea en un archivo **VRML** es con la que el navegador identifica la versión del archivo, es decir, la **cabecera**. Para **VRML 2.0** la cabecera es la siguiente: `#VRML V2.0 utf8`. Donde `#VRML V2.0` denota el tipo y la versión, y `utf8` permite utilizar la codificación **UTF-8** para poder emplear todos los caracteres especiales del estándar **ISO 10646**.

Los comentarios en **VRML** se denotan colocando un signo `#` al comienzo del comentario.

VRML 2.0 es **sensible al contexto**, es decir, una variable en minúsculas es diferente a la misma variable en mayúsculas.

1.4 Nodos, Campos y Eventos

La manipulación de objetos en el mundo virtual se realiza a través de los siguientes objetos definidos en **VRML** [VRML 97]:

- **Nodos. -** son objetos geométricos tridimensionales, imágenes, colores, etc. Algunos de ellos tienen características variables que pueden definirse mediante **campos**, los cuales funcionan como **parámetros**.
- **Campos. -** pueden ser **univaluados** (con un solo valor) o **multivaluados**, cuando se necesita una lista de valores.
- **Eventos. -** son mensajes que circulan entre **nodos** y que permiten la variación de parámetros de un objeto durante la navegación en el entorno virtual. Los eventos pueden ser de **entrada** (si aceptan eventos), de **salida** (si envían eventos) o de **entrada / salida**.

1.5 Unidades de medida

En la **Tabla 1.1** se muestran las unidades convencionales empleadas en **VRML** aunque estas son arbitrarias.

Parámetro	Unidad
Distancias lineales	Metros
Ángulo	Radianes
Tiempo	Segundos
Color	RGB (**Rojo**, **Verde** y **Azul**)

Tabla 1.1: Unidades convencionales usadas en **VRML**.

Para definir un mundo virtual tridimensional (**3D**) en **VRML** se asumen tres ejes de coordenadas **eje x**, **eje y** y **eje z** que representan cada una de las tres dimensiones (**Figura 1.6**).

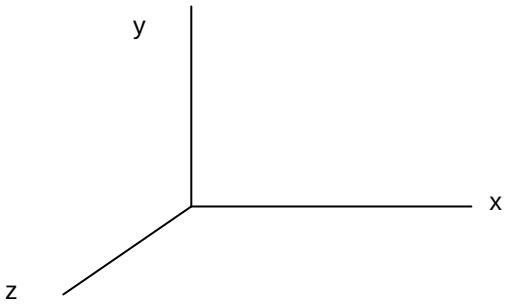


Figura 1.6: Ejes de coordenadas en **VRML**.

Cada objeto en el mundo virtual es desarrollado bajo un **sistema de ejes local**, por lo que para colocarlo dentro del mundo virtual y en una posición diferente al origen, será necesario aplicar una transformación para situarlo en el lugar correspondiente.

1.6 Primitivas y materiales

Existen en **VRML** algunas primitivas que definen una serie de objetos simples. Para poder visualizarlas, hay que usar un nodo `Shape`, el cual presenta dos campos `geometry` y `appearance`. Las primitivas para el campo `geometry` son:

- `Box {size x y z}` para dibujar un **cubo** de dimensiones `x`, `y` y `z` para cada uno de los ejes coordenados.
- `Sphere {radius r}` para dibujar una **esfera** con radio `r`.
- `Cone {bottomRadius r height h}` para dibujar un **cono** con una circunferencia en la base de radio `r` y altura `h`.
- `Cylinder {radius r height h}` para dibujar un **cilindro** con una circunferencia de radio `r` y altura `h`.

Las primitivas **VRML** anteriores definen la geometría de los objetos básicos. También es posible definir un **color** ó alguna **textura**. La **apariencia** de una primitiva se establece por medio del campo `appearance` del nodo `Shape`. Algunas primitivas de este nodo son:

- `Material`.- para definir un material de la primitiva como **color** y **textura**.
- `diffuseColor R G B`.- para establecer el color del material empleado. Se emplea el formato de color **RGB** (**R** para el **rojo**, **G** para el **verde** y **B** para el **azul**). Se tiene que indicar el valor de cada uno de los parámetros **RGB**, que representan la intensidad de cada uno de los colores básicos. En donde cada parámetro varía en el rango de `[0,1]`. En las **Tablas 1.2** y **1.3** es posible observar como se combinan estos parámetros para proporcionar múltiples colores y texturas.

Color	Color RGB
Blanco	1 1 1
Amarillo	1 1 0
Cyan	0 1 1
Verde	0 1 0
Magenta	1 0 1
Rojo	1 0 0
Azul	0 0 1

Tabla 1.2: Colores básicos en **VRML**.

Descripción	AmbientColor	DiffuseColor	SpecularColor	shininess
Dorado	0.57 0.40 0.00	0.22 0.15 0.00	0.71 0.70 0.56	0.16
Aluminio	0.30 0.30 0.35	0.30 0.30 0.50	0.70 0.70 0.80	0.09
Cobre	0.33 0.26 0.23	0.50 0.11 0.00	0.95 0.73 0.00	0.93
Púrpura	0.25 0.17 0.19	0.10 0.03 0.22	0.64 0.00 0.98	0.08
Rojo	0.25 0.15 0.15	0.27 0.00 0.00	0.61 0.13 0.18	0.12
Azul	0.10 0.11 0.79	0.30 0.30 0.71	0.83 0.83 0.83	0.12

Tabla 1.3: Colores metálicos con otros campos de **VRML**.

El **Programa 1.1** muestra un ejemplo sencillo de un programa en **VRML**. Este programa dibuja una caja amarilla con dimensiones en **x** de **4** unidades, en **y** de **3** unidades y en **z** con **5** unidades. Obsérvese que la geometría se define dentro de un nodo **Shape** y la apariencia dentro del nodo que define a la geometría. En la **Figura 1.7** se observa el resultado, al observar la caja desde una esquina.

```
#VRML V2.0 utf8
Shape {
      geometry Box{ size 4 3 5 }          # Las dimensiones son: X Y y Z
      appearance Appearance{
            material Material {
                  diffuseColor 1 1 0
                  }
            }
      }
```

Programa 1.1: Uso de las primitivas básicas de **VRML**.

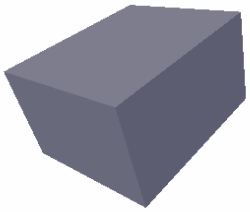


Figura 1.7: Caja simple en **VRML**.

1.7 Texto en VRML

En muchas ocasiones es necesaria la presentación de texto dentro de los ambientes virtuales; **VRML** provee una forma de agregar texto como si se tratase de objetos, a través de la primitiva **Text** del campo **geometry**, en donde también es posible definir otros parámetros de la letra como son el tipo, estilo, tamaño, etc. (**Programa 1.2** y **Figura 1.8**).

```
#VRML V2.0 utf8
Shape {
      geometry Text {                     # Inserta Texto
            string ["VRML"]               # Texto a insertar: VRML
            fontStyle FontStyle {         # Define parámetros de la letra
                  family      "ARIAL"     # Tipo de letra: ARIAL
                  style       "BOLD"      # Estilo: negrita
                  size        1.5         # Define el tamaño: 1.5 Unidades
                  }
            }
      }
```

Programa 1.2: Código **VRML** para insertar texto.

VRML

Figura 1.8: Texto en el espacio virtual con **VRML**.

1.8 Información del mundo

La información del mundo, es la información que el navegador presenta al usuario, como es el **título del mundo** (`title`) y otro tipo de información como: autores, palabras clave, etc. (`info`), por ejemplo tenemos el **Programa 1.3**, que simplemente agrega información al mundo.

```
#VRML V2.0 utf8
# Agrega información al mundo virtual
WorldInfo   {
      title "Título que se presenta cuando es ejecutado por el navegador"
      info  "Cualquier tipo de información"
      }
```

Programa 1.3: Información del mundo virtual.

1.9 Fondo

VRML permite colocar un color de fondo, un color degradado en el cielo, poner tierra ó poner imágenes como escenografía.

Para colocar un color de fondo como sólido ó color degradado, **VRML** emplea el concepto de esfera de radio infinito que engloba a todo el mundo. La tierra es la semiesfera boca arriba de la parte inferior del mundo. Para la manipulación de la esfera se emplea el nodo `Background`.

Para definir **un solo color del cielo**, se hace empleando el campo `skyColor` de la forma en que se muestra en el **Programa 1.4** (**Figura 1.9**).

```
#VRML V2.0 utf8
# Fondo
Background {
      skyColor [ .8 .8 1 ]
      }
# Una esfera dentro del mundo
Shape {
      geometry Sphere { radius 2 }
      appearance Appearance {
            material Material {diffuseColor 0 .82 .98}
            }
      }
```

Programa 1.4: Definir un solo color del cielo.

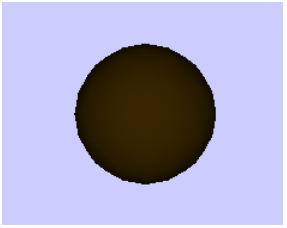


Figura 1.9: Un solo color del cielo como fondo.

Para definir un fondo con **color de cielo degradado**, se hace (**Programa 1.5** y **Figura 1.10**):

```
#VRML V2.0 utf8
# Fondo
Background {
      skyAngle [         .384 , .785  , 1.047    , 1.309  , 1.484  , 1.5708 ]
      skyColor [ 0 0 .2, 0 0 1, 0 1 1 , .75 .75 1, .8 .8 0, .8 .6 0, 1 .4 0 ]
      }
# Una esfera dentro del mundo
Shape {
      geometry Sphere { radius 2 }
      appearance Appearance {
            material Material {diffuseColor 0 .82 .98}
            }
      }
```

Programa 1.5: Definir colores del cielo degradados.

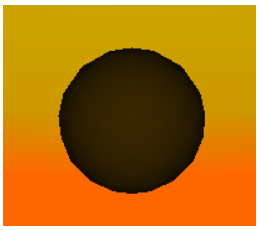


Figura 1.10: Cielo con colores degradados como fondo.

Para definir un fondo con **color de cielo degradado** y **color de tierra**, se muestra el **Programa 1.6** y la **Figura 1.11**.

```
#VRML V2.0 utf8
# Fondo
Background {
      skyAngle    [         .384 , .785  , 1.047    , 1.309  , 1.484  , 1.5708 ]
      skyColor    [ 0 0 .2, 0 0 1, 0 1 1 , .75 .75 1, .8 .8 0, .8 .6 0, 1 .4 0 ]
      groundAngle [           1.5708   ]
      groundColor [ .1 .8 .2, .1 .8 .2 ]
      }
# Una esfera dentro del mundo
Shape {
      geometry Sphere { radius 2 }
      appearance Appearance {
            material Material {diffuseColor 0 .82 .98}
            }
      }
```

Programa 1.6: Definir colores del cielo degradados y color de tierra.

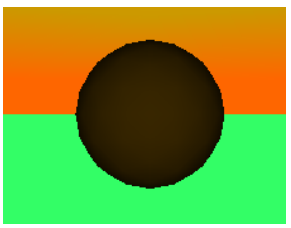


Figura 1.11: Colores del cielo degradados y color de tierra como fondo.

Para definir un **fondo con seis imágenes alrededor** se tiene el **Programa 1.7** y la **Figura 1.12**.

```
#VRML V2.0 utf8
# Fondo
Background {
      frontUrl    "frente.jpg"
      backUrl     "atras.jpg"
      topUrl      "arriba.jpg"
      bottomUrl   "tierra.jpg"
      rightUrl    "derecha.jpg"
      leftUrl     "izquierda.jpg"
      }
# Una esfera dentro del mundo
Shape {
      geometry Sphere { radius 2 }
      appearance Appearance {
            material Material {diffuseColor 0 .82 .98}
            }
      }
```

Programa 1.7: Fondo con seis imágenes alrededor.

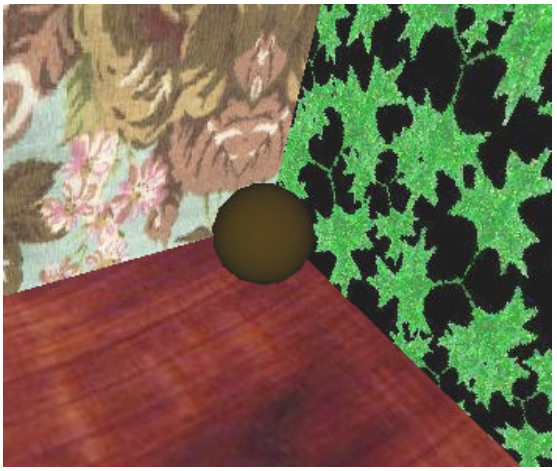


Figura 1.12: Fondo con seis imágenes alrededor.

Capítulo 2: Transformaciones

Para poder ubicar los objetos en un mundo virtual en alguna posición deseada, es necesario aplicar algunas **transformaciones** [VRML 97]. Las transformaciones básicas son la translación, la rotación y la escala.

2.1 Transformaciones en VRML

Las **transformaciones** en **VRML** se emplean para situar, escalar y orientar los objetos dentro del mundo virtual.

La **translación** se define con un nodo `translation X Y Z`, que define el desplazamiento del objeto en cada una de las coordenadas. Al nodo `translation` le sigue un nodo `children` el cual contiene la lista de objetos que serán afectados por la transformación definida por el nodo `Transform`.

Por ejemplo, en el siguiente programa (**Programa 2.1** y **Figura 2.1**), se muestra una esfera en el centro del mundo virtual y otra desplazada `10 unidades` en los ejes `X` y `Z`, respecto a la primera.

```
#VRML V2.0 utf8
Shape {                     # Esfera de referencia en el centro
      geometry Sphere{ radius 2 }
      appearance Appearance{
            material Material {diffuseColor 1 0 0}
            }
      }
Transform {
      translation 10 0 10     # Desplazamiento en X y Z
      children [              # Objetos a desplazar
            Shape {
                  geometry Sphere{ radius 2 }
                  appearance Appearance{
                        material Material {diffuseColor 1 1 0 }
                        }
                  }
            ]
      }
```

Programa 2.1: Nodo `translation`.

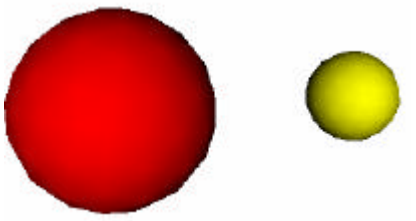


Figura 2.1: Nodo `translation`.

La **rotación** se define con el nodo `rotation X Y Z a`, en donde `X Y` y `Z` definen el vector en el espacio tridimensional de un objeto ó la orientación del eje de rotación, y `a` es el ángulo de rotación en radianes alrededor del eje definido anteriormente. Se emplea un nodo `children` que contiene los objetos que serán afectados por la rotación.

Un ejemplo de la forma en que son empleados ambos nodos se tiene en el **Programa 2.2** y la **Figura 2.2**, donde se tiene una caja en el origen y dentro de ésta, otra caja con una rotación en el **eje Y** de **45º** (**0.7854 rad**).

```
#VRML V2.0 utf8
Shape {                              # Caja de referencia en el Centro
      geometry Box{ size 4 2 4 }     # Las dimensiones son: X Y y Z
      appearance Appearance{
            material Material {diffuseColor 1 1 0}
            }
      }
Transform {
      rotation 0 1 0 .7854           # Eje y ángulo de rotación
      children [
            Shape{                   # Contiene  los objetos a transformar
                  geometry Box {size 2 4 2 }
                        appearance Appearance {
                              material Material {diffuseColor 0 0 1}
                              }
                        }
                  ]
            }
```

Programa 2.2: Nodo **`rotation`**.

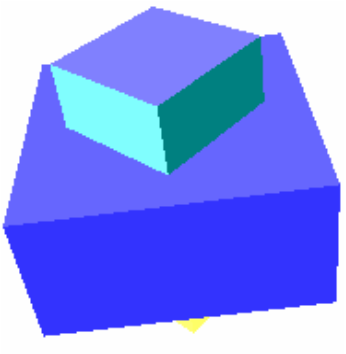


Figura 2.2: Nodo **`rotation`**.

El **escalamiento** de un objeto es de dos tipos: **escalado uniforme** y **escalado no uniforme**. El primero cambia el tamaño del objeto en todas las direcciones, así los objetos mantienen sus proporciones. El **escalado no uniforme** cambia el tamaño de un objeto sólo en una dirección, deformando al objeto.

Un ejemplo de un **escalado no uniforme** se muestra en el **Programa 2.3** y la **Figura 2.3**. Donde se tiene una caja con dimensiones unitarias, escalada en el **`eje X`** por **2** y en el **`eje Z`** por **3**.

```
#VRML V2.0 utf8
Transform {
      scale 2 1 3                    # Define la escala en X Y y Z
      children [
            Shape {
                  geometry Box{ size 1 1 1 }
                  appearance Appearance{
                        material Material {diffuseColor 0 1 0 }
                        }
                  }
            ]
      }
```

Programa 2.3: Nodo **`scale`**.

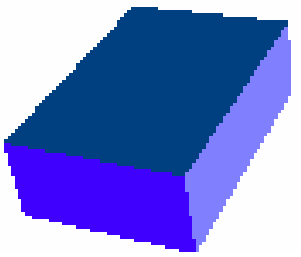


Figura 2.3: Nodo `scale`.

2.2 Encadenamiento de transformaciones

Cuando se desea emplear más de una transformación, no es necesario definir un nodo para cada una, todas se pueden realizar en el mismo nodo.

El orden en que se aplican las transformaciones, es de adentro hacia afuera. En el **Programa 2.4**, se observa una caja que primero se traslada, luego se rota y por último se le aplica una escala (**Figura 2.4**).

```
#VRML V2.0 utf8
Shape {
      geometry Sphere{radius .2}         # Esfera como referencia
      appearance Appearance{ material Material{ diffuseColor 1 0 0}}
      }

Transform{
      scale .5 .5 1
      children [
            Transform {
                  rotation 1 0 0 .5236
                  children [
                        Transform {
                              translation 5 4 0
                              children [
                                    Shape{geometry Box {size 1 1 1}
                                    appearance Appearance{
                                          material Material{
                                                diffuseColor 0 .5 .2}}
                                          }
                                    ]
                              }
                        ]
                  }
            ]
      }
```

Programa 2.4: Encadenamiento de transformaciones.

Figura 2.4: Resultados de un encadenamiento de transformaciones.

2.3 Puntos de vista con VRML

VRML permite definir diversos **puntos de vista**. Los usuarios pueden utilizarlos para moverse a partes importantes dentro del mundo virtual. Esta operación se realiza mediante el nodo **`Viewpoint`**.

El nodo **`Viewpoint`** se utiliza de la forma:

```
Viewpoint {
      position    X1 Y1 Z1
      orientation X2 Y2 Z2 α
      description "Nombre"
      }
```

Donde **X_1**, **Y_1** y **Z_1** son el **vector de la posición** donde se sitúa el punto de vista, luego **X_2**, **Y_2** y **Z_2** definen el **eje de rotación** del punto de vista en un **ángulo** a, y **`Nombre`** designa una **etiqueta** para identificar el punto de vista, este aparece en el navegador.

Cuando se ingresa al navegador, el punto de vista por omisión es:

```
Viewpoint {
      position    0 0 10
      orientation 0 0 1 0
      description "Punto de Vista Inicial"
      }
```

También es posible definir un **ángulo de visión** b, para que sólo sea visible una determinada área, se emplea el nodo **`fieldOfView`** de la forma:

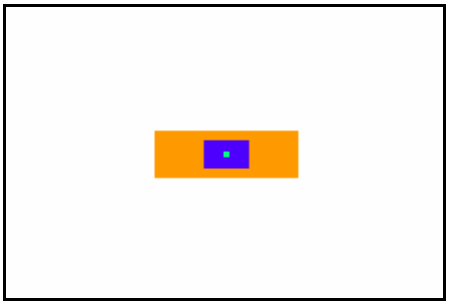
```
Viewpoint {
      fieldOfView β            # β es el ángulo de abertura
      description "Nombre"
      }
```

El paso de un punto de vista a otro es suave, sin embargo si se requiere pasar de un punto de vista a otro de forma instantánea, es necesario modificar el campo **`jump`** y ponerlo a **`TRUE`**. Si se desea pasar de forma suave es necesario modificar el campo **`jump`** y ponerlo a **`FALSE`**.

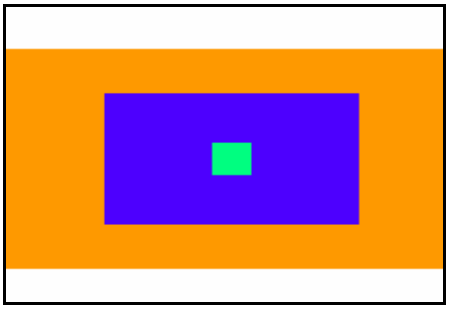
En el **Programa 2.5**, se observa el uso de diversos puntos de vista para un mundo sencillo, y en la **Figura 2.5** se observan los resultados.

```
#VRML V2.0 utf8
# Punto de vista 1
Viewpoint {
      fieldOfView 2.618
      description "Gran Angular"
      }

# Punto de vista 2
Viewpoint {
      fieldOfView 1.3962667
      description "Normal"
      }

# Punto de vista 3
Viewpoint {
      fieldOfView .31416
      description "Telefoto"
      }

# Punto de vista 4
Viewpoint {
      position    0 0 10
      orientation 0 0 1 0
      description "Punto de Vista por Defecto"
      }

# Punto de vista 5
Viewpoint {
      jump        TRUE
      position    0 10 10
      orientation 1 0 0 0
      description "Punto de Vista con Salto"
      }

# Caja 1
Shape {
      geometry Box { size 30 10 2 }
      appearance Appearance {
            material Material {diffuseColor 1 .6 0}
            }
      }

# Caja 2
Shape {
      geometry Box { size 8 5 5 }
      appearance Appearance {
            material Material {diffuseColor .3 0 1}
            }
      }

# Caja 3
Shape {
      geometry Box { size 1 1 8 }
      appearance Appearance {
            material Material {diffuseColor 0 1 .5}
            }
      }
```

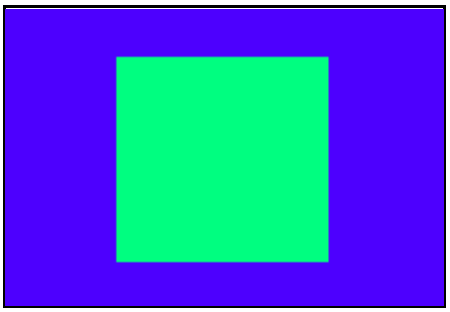
Programa 2.5: Distintos puntos de vista.



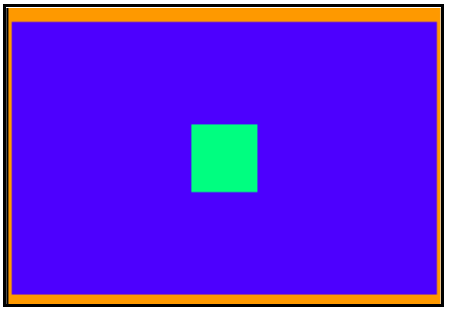
a) Gran Angular.



b) Normal.



c) Telefoto.



d) Por Omisión.

Figura 2.5: Ejemplos de **puntos de vista** en un mundo virtual simple.

Capítulo 3: Puntos, Líneas, Objetos y Mallas

Las primitivas básicas permiten modelar sólo unas cuantas formas geométricas, sin embargo **VRML** permite definir **puntos**, **líneas** y **objetos** a través de los nodos: **PointSet**, **IndexedLineSet** e **IndexedFaceSet [VRML 97]**.

3.1 Nodo PointSet

El nodo **PointSet** se emplea para definir **puntos aislados** en el espacio, que conforman un sólo objeto.

```
geometry PointSet{
      coord Coordinate{
            # Lista de vértices
            point [
                  x0 y0 z0, x1 y1 z1,
                  .........
                  x n-1 y n-1 zn-1, xn yn zn
                  ]
            }
      }
```

En la **Figura 3.1**, se define una pirámide en base a sus vértices.

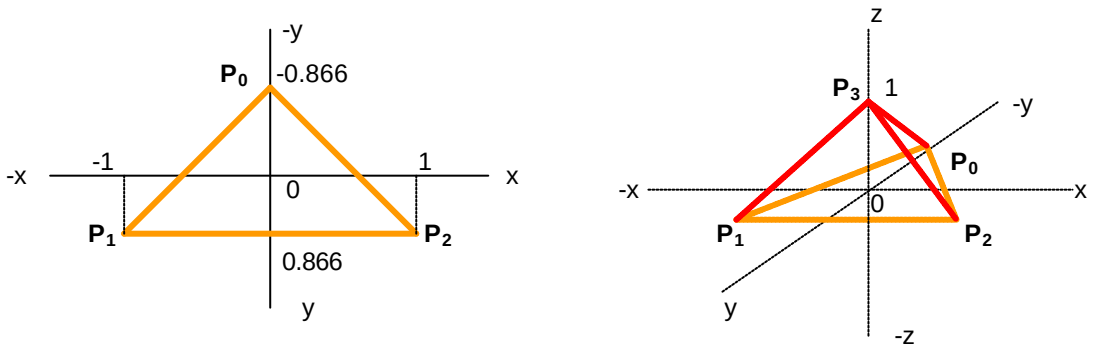


Figura 3.1: Definición de los vértices de una pirámide con **VRML**.

Por ejemplo, de la **Figura 3.1**, es posible determinar los puntos necesarios para definir una pirámide, como se muestra en el **Programa 3.1** y en la **Figura 3.2**.

```
#VRML V2.0 utf8
Shape       {
      geometry PointSet {
            coord Coordinate{
                  point [
                        0    -.866 0,      # Punto P0
                        -1   .866  0,      # Punto P1
                        1    .866  0,      # Punto P2
                        0    0     1       # Punto P3

                        ]
                  }
            }
      }
```

Programa 3.1: Nodo **PointSet**.

3.2 Nodo IndexedLineSet

El nodo **IndexedLineSet** se emplea para definir objetos mediante sus aristas para crear una **figura de alambre** , es decir, los puntos se enlazan mediante **líneas**. El nodo **IndexedLineSet** se define de la forma:

```
geometry IndexedLineSet {
       coord Coordinate{
            # Lista de vértices
            point [
                  x0 y0 z0,
                  x1 y1 z1,
                  .........
                  .........
                  x n-1 y n-1 zn-1,
                  xn yn zn
                  ]
            }
      # Lista de puntos enlazados por líneas
      coordIndex [
            0,1,2,...,-1,
            3,4,5,...,-1,
            ............
            n-2,n-1,n,-1
            ]
      }
```

De la **Figura 3.1**, podemos establecer los puntos y las líneas que se desea aparezcan en el mundo virtual, como se observa en el **Programa 3.2** y en la **Figura 3.2**.

```
#VRML V2.0 utf8
Shape       {
      geometry IndexedLineSet {
            coord Coordinate{
                  point [
                        0    -.866 0,      # Punto P0
                        -1   .866  0,      # Punto P1
                        1    .866  0,      # Punto P2
                        0    0     1       # Punto P3
                        ]
                  }
            coordIndex [
                  0,1,2,0,-1,              # Líneas de la base
                  0,3,-1,                  # P0 a P3
                  1,3,-1,                  # P1 a P3
                  2,3,-1                   # P2 a P3
                  ]
            }
      }
```

Programa 3.2: Nodo **IndexedLineSet**.

3.3 Nodo IndexedFaceSet

El modelado de objetos mediante el nodo **IndexedFaceSet**, se realiza definiendo los **vértices** y **caras del objeto** a través de arreglos de coordenadas de puntos y caras, de la forma siguiente:

```
geometry IndexedFaceSet{
      coord Coordinate{
            # Lista de vértices
            point [
                  x0 y0 z0, x1 y1 z1,
                  .........
                  x n-1 y n-1 zn-1, xn yn zn
                  ]
            }
      coordIndex [
            # Lista de caras
            0,1,2,...,-1, 3,4,5,...,-1,
            ............, n-2,n-1,n,-1
            ]
      }
```

Se debe tener en cuenta las reglas siguientes:

- Los **vértices** tienen que ser consecutivos, siguiendo el perímetro de la cara.
- Los **vértices** se colocan en un orden contrario al de la dirección de las manecillas del reloj, pues la **iluminación** se refleja en un vector normal, hacia fuera de ésta cara. Mirando desde el lado contrario, la cara es transparente.
- El **último vértice** de cada cara se enlaza con el primero, para formar un perímetro cerrado.

Con los puntos anteriores, no es difícil, definir los vértices y posteriormente las caras, en el **Programa 3.3** y en la **Figura 3.2** se muestra el uso del nodo **IndexedFaceSet**.

```
#VRML V2.0 utf8
Shape {
      geometry IndexedFaceSet{
            coord Coordinate{
                  point [
                        0   -.866 0,      # Punto P0
                        -1  .866  0,      # Punto P1
                        1   .866  0,      # Punto P2
                        0   0     1       # Punto P3
                        ]
                  }
            coordIndex [
                  0,2,1,-1,               # Base iluminada abajo
                  0,3,2,-1,               # Cara 1
                  0,1,3,-1,               # Cara 2
                  1,2,3,-1,               # Cara 3
                  ]
            }
      }
```

Programa 3.3: Nodo **IndexedFaceSet**.

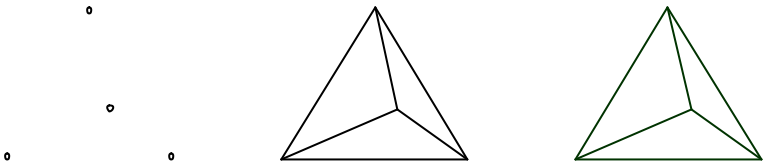


Figura 3.2: Pirámide definida con Puntos, Líneas y como Objeto.

Las caras quedan en color blanco, sin embargo es posible **modificar los colores** modificando el nodo y definiendo el color para cada cara en el nodo **`colorPerVertex`**, de la forma en que se muestra en el **Programa 3.4** (**Figura 3.3**).

```
#VRML V2.0 utf8
Shape {
      geometry IndexedFaceSet{
            coord Coordinate{
                  point [
                        0   -.866 0,      # Punto P0
                        -1  .866  0,      # Punto P1
                        1   .866  0,      # Punto P2
                        0   0     1       # Punto P3
                        ]
                  }
            coordIndex [
                  0,2,1,-1,               # Base iluminada abajo
                  0,3,2,-1,               # Cara 1
                  0,1,3,-1,               # Cara 2
                  1,2,3,-1,               # Cara 3
                  ]
            colorPerVertex FALSE
            color Color {
                  color [
                        1  1  0,          # Base Amarilla
                        0  0  1,          # Cara 1 Azul
                        0  1  0,          # Cara 2 Verde
                        1  0  0           # Cara 3 Roja
                        ]
                  }
            }
      }
```

Programa 3.4: Nodo **`IndexedFaceSet`**, con modificación de color.

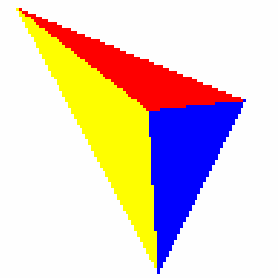


Figura 3.3: Pirámide con caras de colores.

3.4 Mallas

Las mallas se emplean para describir valles, montañas, figuras complejas, etc. Se puede usar el nodo **IndexedFaceSet**, para definir una figura de este tipo; sin embargo, una forma más eficiente de realizar figuras complejas es usar **ElevationGrid**.

El nodo **ElevationGrid** se crea con las siguientes características:

- **Dimensiones X Y Z** (**xDimension**, **yDimension** y **zDimension**).- tamaño de la malla, como un arreglo de dos dimensiones (sólo se define en dos ejes).
- **Espacios X Y Z** (**xSpacing**, **ySpacing** y **zSpacing**).- distancias entre cada fila y cada columna (sólo se define en dos dimensiones).
- **Altura** (**height**).- puntos de elevación (en el eje no definido anteriormente), se colocan como valores en el arreglo.
- **Sólido** (**solid [FALSE,TRUE]**).- permite que la malla sea sólida ó no.

En la **Figura 3.4** se observa la forma de definir una malla en **VRML**.

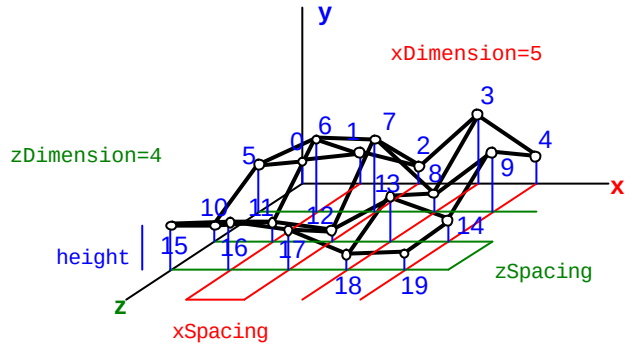


Figura 3.4: Mallas en **VRML**.

En el **Programa 3.5**, se observa un ejemplo de una montaña modelada como malla, en donde además se le asigna un color (verde) a todas las caras que la conforman (**Figura 3.5**).

```
#VRML V2.0 utf8
Shape {
      appearance Appearance { material Material { diffuseColor 0 1 0 } }
      geometry ElevationGrid {
            xDimension  9
            zDimension  9
            xSpacing    1.0
            zSpacing    1.0
            solid       FALSE
            height [    0.0, 0.0, 0.5, 1.0, 0.5, 0.0, 0.0, 0.0, 0.0,
                        0.0, 0.0, 0.0, 0.0, 2.5, 0.5, 0.0, 0.0, 0.0,
                        0.0, 0.0, 0.5, 0.5, 3.0, 1.0, 0.5, 0.0, 1.0,
                        0.0, 0.0, 0.5, 2.0, 4.5, 2.5, 1.0, 1.5, 0.5,
                        1.0, 2.5, 3.0, 4.5, 5.5, 3.5, 3.0, 1.0, 0.0,
                        0.5, 2.0, 2.0, 2.5, 3.5, 4.0, 2.0, 0.5, 0.0,
                        0.0, 0.0, 0.5, 1.5, 1.0, 2.0, 3.0, 1.5, 0.0,
                        0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2.0, 1.5, 0.5,
                        0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5, 0.0, 0.0
                  ]
            }
      }
```

Programa 3.5: Modelación de una montaña como malla.

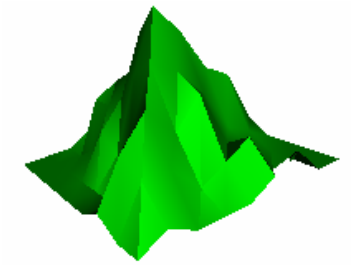


Figura 3.5: Vista de la montaña modelada.

3.5 Extrusión

Un tubo, una barra, un vaso, trazos de partículas, líneas de flujo, etc. pueden ser modelados con facilidad usando el nodo **`Extrusion`**. Una figura por extrusión, se define por:

- Una **sección de cruce** en dos dimensiones (**`crossSection`**).
- Una **línea en tres dimensiones** (**`spine`**) a lo largo de la extensión de la sección de cruce.

La estructura del nodo **`Extrusion`** es la siguiente:

```
Extrusion {
    crossSection [ . . . ]
    spine [ . . .  ]
    . . .
    scale [ . . . ]
    orientation [ . . . ]
}
```

Por ejemplo, tenemos el **Programa 3.6**, en donde se observa el modelo de una caja torcida (**Figura 3.6**).

```
#VRML V2.0 utf8
Shape {
      appearance Appearance {
            material Material { diffuseColor 1.0 0.5 0.0 }
            }
      geometry Extrusion {
            creaseAngle 0.785
            crossSection [
                  -1  1,  1  1,
                   1 -1, -1 -1,
                  -1  1
                  ]
            spine [
                  0 0   0, 0 0.5 0, 0 1.0 0,
                  0 1.5 0, 0 2.0 0, 0 2.5 0,
                  0 3.0 0, 0 3.5 0, 0 4.0 0
                  ]
            orientation [
                  0 1 0 0.0,  0 1 0 0.175,0 1 0 0.349,
                  0 1 0 0.524,0 1 0 0.698,0 1 0 0.873,
                  0 1 0 1.047,0 1 0 1.222,0 1 0 1.396
                  ]
            }
      }
```

Programa 3.6: Nodo **`Extrusion`**.

Figura 3.6: Figura definida con el nodo **Extrusion**.

Otro ejemplo lo tenemos en el **Programa 3.7**, en donde se observa el modelo de un corazón realizado con una extrusión (**Figura 3.7**).

```
Shape {
      geometry Extrusion {
            creaseAngle 3.14
            crossSection [    0  0.8, 0.2     1,
                              0.7  1, 1     0.5,
                              0.8  0, 0.5  -0.3,
                              0 -0.7, -0.5 -0.3,
                              -0.8 0, -1    0.5,
                              -0.7 1, -0.2    1, 0 0.8
                        ]
            scale         [   0.01 0.01, 0.85 0.85,
                              1       1, 0.85 0.85,
                              0.01 0.01
                        ]
            solid FALSE
            spine         [   0  0  0, 0 0.1 0,
                              0 0.5 0, 0 0.9 0, 0 1 0
                        ]
            }
      appearance Appearance {
            material Material { diffuseColor 0.8 0.3 0.3 }
            }
      }
```

Programa 3.7: Corazón con el nodo **Extrusion**.

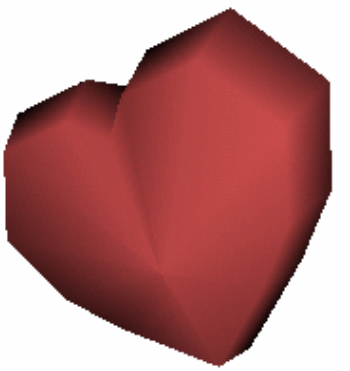


Figura 3.7: Corazón con el nodo **Extrusion**.

Un ejemplo más se muestra en el **Programa 3.8**, en donde se observa el modelo de un marco por extrusión (**Figura 3.7**).

```
#VRML V2.0 utf8
# Figura por extrusión
Shape {
      geometry Extrusion {
            creaseAngle 3.14
            crossSection [    0  0,
                              0  1,
                              1  1,
                              0  0
                        ]
            solid FALSE
            spine       [     0 0 0,
                              0 3 0,
                              0 3 3,
                              0 0 3,
                              0 0 0
                        ]
            }
      appearance Appearance {
            material Material {
                  diffuseColor      .8 .63 0
                  specularColor     .5 .5 .5
                  ambientIntensity  0
                  emissiveColor     .14 .11 0
                  }
            }
      }

# Caja de fondo del marco
      Transform {
            rotation    0 1 0 1.57
            translation 0 1.5 1.5
            children    Shape {
            geometry Box {
                  size  3 3 .01
                  }
      }
}
```

Programa 3.8: Marco con el nodo **`Extrusion`**.

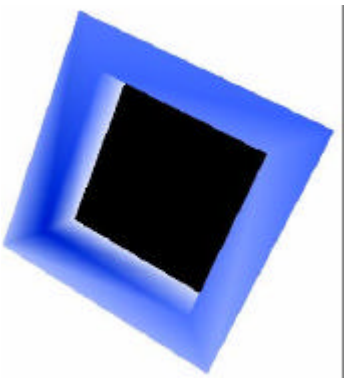


Figura 3.8: Marco con el nodo **`Extrusion`**.

Capítulo 4: Iluminación y Texturas

En **VRML**, existen tres formas de iluminación: luz direccional, luz puntual y luz spot, cada una corresponde a las formas típicas de iluminación que es posible realizar a través de la computadora **[VRML 97]**.

4.1 Iluminación ambiente

En **VRML**, se puede controlar la **luz ambiente** para cada tipo de luz y el **control de la reflexión** para cada material. El nodo **`Material`**, tiene el campo **`ambientIntensity`**, el cuál determina en que proporción el material refleja la intensidad ambiente de las luces que se encuentren definidas en el entorno. Su valor por omisión es **0.2**, su rango es **`[0,1]`**.

4.2 Luz direccional

Con el nodo **`DirectionalLight`**, se pueden iluminar objetos como si fueran rayos del sol al llegar a la Tierra, es decir, todos los rayos son paralelos entre ellos y por lo tanto inciden en el mismo ángulo sobre la superficie.

Antes de realizar una prueba de iluminación es necesario apagar una luz que existe por defecto, conocida como **`headlight`**, ó **luz de casco de minero**, se puede apagar desde la información del navegador de la forma: **`NavigationInfo { headlight FALSE }`**.

La forma en que se usa el nodo **`DirectionalLight`**, se muestra en el **Programa 4.1**, en donde se muestran tres fuentes de luz que iluminan una esfera (**Figura 4.1**).

```
#VRML V2.0 utf8
NavigationInfo { headlight FALSE } # Apaga la luz de Casco de Minero
DirectionalLight {                 # Luz roja
      ambientIntensity  0.5
      color             1 0 0
      intensity         1          # Intensidad de la luz ambiente
      direction         0.5 -0.5 -0.1
      }
DirectionalLight {                 # Luz verde
      ambientIntensity  0.5
      color             0 1 0
      intensity         1
      direction         -0.5 -0.5 -0.1
      }
DirectionalLight {                 # Luz 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 }  # Esfera a iluminar
      }
```

Programa 4.1: Iluminación direccional.

Figura 4.1: Resultados de la **iluminación direccional**.

Cada haz de luz incide sobre la esfera en el ángulo especificado, sin embargo los colores en algunos lugares de la esfera se combinan, tal y como ocurre en un ambiente real.

4.3 Luz puntual

Una **luz puntual** tiene la propiedad de irradiar **rayos de luz en todas direcciones** a partir de un punto dado en el espacio. En estas fuentes de luz, la luz se atenúa conforme se aleja de la fuente luminosa.

La **atenuación** es una propiedad física que se manifiesta debido a la pérdida de energía de los fotones a medida que se alejan de la fuente de luz. Se controla a través del nodo **`PointLight`**, en el campo **`attenuation`**.

Un ejemplo se muestra en el **Programa 4.2** en donde se ilumina una figura blanca con una luz cyan, el resultado se observa en la **Figura 4.2**

```
#VRML V2.0 utf8

NavigationInfo { headlight FALSE } # Apaga la luz de Casco de Minero
PointLight {                       # Luz cyan con atenuación
      location          8 -3 -10
      radius            25
      ambientIntensity  0.5
      color             0 1 1
      intensity         3
      attenuation       0 0.4 0     # Posee algo de atenuación
      }

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

Programa 4.2: Uso del nodo **`PointLight`**.

Figura 4.2: Uso del nodo **`PointLight`**.

4.4 Luz Spot

La **luz Spot** es un tipo de iluminación compleja, en donde se debe de especificar la dirección y el ángulo de apertura, para dar la sensación de una **iluminación de teatro**. Un ejemplo de este tipo de iluminación se tiene en el **Programa 4.3** y en la **Figura 4.3**.

```
#VRML V2.0 utf8
NavigationInfo { headlight FALSE }       # Apaga la luz de Casco de Minero
SpotLight {
      location          0 0 -10
      radius            30
      ambientIntensity  0.5
      color             1 0 0
      intensity         1
      attenuation       1 0 0
      direction         0 -1 0          # Apunta hacia abajo
      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
                        ]
                  }
            }
      }
```

Programa 4.3: Uso de la **luz Spot** para iluminar una superficie plana.

Figura 4.3: Uso de la **luz Spot** para iluminar una superficie plana.

El campo **direction** permite dar el vector de **dirección** de luz. El campo **cutOffAngle** determina el **ángulo de apertura** total del haz de luz, hace referencia a la mitad del ángulo total de apertura. El campo **beamWidth** define un **ángulo interno** donde la intensidad del haz de luz es máxima y uniforme.

Otro ejemplo con **luz Spot** se muestra en el **Programa 4.4** y la **Figura 4.4**.

```
#VRML V2.0 utf8
NavigationInfo { headlight FALSE } # Apaga la luz de Casco de Minero
SpotLight { location 0 10 -15
      radius            30
      ambientIntensity  0.5
      color             0 1 0
      intensity         1
      attenuation       1 0 0
      direction         0 -1 0      # Apunta hacia abajo
      cutOffAngle       1.0472      # 60 grados en radianes
      beamWidth         0.785398    # 45 grados en radianes
      }
Transform {
      translation       0 0 -10
      children Shape {
            appearance Appearance {
                  material Material {
                        ambientIntensity  1
                        diffuseColor      1 1 1
                        }
                  }
            geometry ElevationGrid {
                  xDimension  9
                  zDimension  9
                  xSpacing    1.0
                  zSpacing    1.0
                  solid       FALSE
                  height [
                              0.0, 0.0, 0.5, 1.0, 0.5, 0.0, 0.0, 0.0, 0.0,
                              0.0, 0.0, 0.0, 0.0, 2.5, 0.5, 0.0, 0.0, 0.0,
                              0.0, 0.0, 0.5, 0.5, 3.0, 1.0, 0.5, 0.0, 1.0,
                              0.0, 0.0, 0.5, 2.0, 4.5, 2.5, 1.0, 1.5, 0.5,
                              1.0, 2.5, 3.0, 4.5, 5.5, 3.5, 3.0, 1.0, 0.0,
                              0.5, 2.0, 2.0, 2.5, 3.5, 4.0, 2.0, 0.5, 0.0,
                              0.0, 0.0, 0.5, 1.5, 1.0, 2.0, 3.0, 1.5, 0.0,
                              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2.0, 1.5, 0.5,
                              0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5, 0.0, 0.0
                        ]
                  }
            }
      }
```

Programa 4.4: Montaña iluminada con **luz Spot**.

Figura 4.4: Montaña iluminada con **luz Spot**.

4.5 Materiales brillantes

En el nodo **Material** se pueden controlar las características de **brillo** de un material, a través de los campos **specularColor** y **shininess**. El primero define el color que presenta el objeto al reflejar especularmente la luz. El segundo determina como definido o borroso el reflejo del objeto.

En el siguiente ejemplo (**Programa 4.5** y **Figura 4.5**) se observan cinco esferas, en donde es posible apreciar algunas combinaciones de estos parámetros.

```
#VRML V2.0 utf8
Shape {
# Objeto de referencia con material mate
      geometry Sphere { radius 1.1 }
      appearance Appearance {
            material Material {
                  diffuseColor            0.5 0.5 0.5
                  }
            }
      }
Transform {
      translation                         -2 2 0
      children Shape {
            geometry Sphere { radius 1.1 }
            appearance Appearance {
                  material Material {
                        diffuseColor      0.5 0.5 0.5
                        specularColor     1 0 0
                        shininess         0
                        }
                  }
            }
      }
Transform {
      translation                         2 2 0
      children Shape {
            geometry Sphere { radius 1.1 }
            appearance Appearance {
                  material Material {
                        diffuseColor      0.5 0.5 0.5
                        specularColor     1 0 0
                        shininess         0.33
                        }
                  }
            }
      }
```

Programa 4.5: Objetos brillantes (Parte **1**/2).

```
Transform {
      translation                       -2 -2 0
      children Shape {
            geometry Sphere { radius 1.1 }
            appearance Appearance {
                  material Material {
                        diffuseColor      0.5 0.5 0.5
                        specularColor     1 0 0
                        shininess         0.66
                        }
                  }
            }
      }
Transform {
      translation                       2 -2 0
      children Shape {
            geometry Sphere { radius 1.1 }
            appearance Appearance {
                  material Material {
                        diffuseColor      0.5 0.5 0.5
                        specularColor     1 0 0
                        shininess         1
                        }
                  }
            }
      }
```

Programa 4.5: Objetos brillantes (Parte **2**/2).

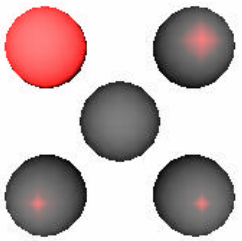


Figura 4.5: Objetos brillantes.

4.6 Objetos autoiluminados

Cuando es necesario que un objeto produzca su propia luz, se usa el campo **`emissiveColor`**, el cual define el color que el material emite sin necesidad de que exista alguna fuente que lo ilumine (**Programa 4.6** y **Figura 4.6**).

```
#VRML V2.0 utf8
NavigationInfo { headlight FALSE }
Shape       {           # Objeto de Referencia
            geometry Cylinder { radius 0.2 height 8 }
            appearance Appearance {
                  material Material {
                        diffuseColor      1 1 1
                        }
                  }
            }
```

Programa 4.6: Objetos **autoiluminados** (Parte **1**/2).

```
Transform {             # Objeto Autoiluminado
      translation                       -0.5 0 0
      children Shape {
            geometry Cylinder { radius 0.2 height 8 }
            appearance Appearance {
                  material Material {
                        diffuseColor      0.5 0.5 0.5
                        emissiveColor     0 0 1
                        }
                  }
            }
      }
```

Programa 4.6: Objetos **autoiluminados** (Parte **2**/2).

Figura 4.6: Objetos **autoiluminados**.

4.7 Materiales transparentes

La **transparencia** de un objeto es otra propiedad que se puede modificar mediante el campo **transparency** del nodo **Material**, como se observa en el **Programa 4.7** y la **Figura 4.7**.

```
#VRML V2.0 utf8
Transform {       # Objeto de fondo de referencia
      translation                       0 0 -2
      children Shape {
            geometry Box { size         20 10 0.5 }
            appearance Appearance {
                  material Material {
                        diffuseColor      1 1 1
                        }
                  }
            }
      }
Transform {
      translation                       -3.75 0 0
      children Shape {
            geometry Box { size         2 4 0.5 }
            appearance Appearance {
                  material Material {
                        diffuseColor      0.5 0.5 0.5
                        transparency      0
                        }
                  }
            }
      }
Transform {
      translation                       -1.25 0 0
      children Shape {
            geometry Box { size         2 4 0.5 }
            appearance Appearance {
```

Programa 4.7: Transparencia en objetos (Parte **1**/2).

```
                material Material {
                      diffuseColor      0.5 0.5 0.5
                      transparency      0.33
                      }
                }
          }
    }
Transform {
    translation                         1.25 0 0
    children Shape {
          geometry Box { size           2 4 0.5 }
          appearance Appearance {
                material Material {
                      diffuseColor      0.5 0.5 0.5
                      transparency      0.66
                      }
                }
          }
    }
Transform {
    translation                         3.75 0 0
    children Shape {
          geometry Box { size           2 4 0.5 }
          appearance Appearance {
                material Material {
                      diffuseColor      0.5 0.5 0.5
                      transparency      0.9
                      }
                }
          }
    }
```

Programa 4.7: Transparencia en objetos (Parte **2**/2).

Figura 4.7: Transparencia en objetos.

4.8 Texturas

El **texturado de objetos** se hace mediante el nodo **appearance**, a través del campo **texture**. Es posible insertar tres nodos: **ImageTexture**, **MovieTexture** y **PixelTexture**. A continuación analizaremos el primero.

Nodo ImageTexture

Este nodo se define en una **URL** (una dirección) de la imagen que ha de servir como textura. La especificación de **VRML** dice que los formatos estándares son: **JPG** y **PNG**, y recomienda que también se acepte el formato **GIF** incluyendo transparencia (**Programa 4.8** y **Figura 4.8**).

```
#VRML V2.0 utf8

Transform {
      translation                  -2 0 0
      children Shape {
            geometry Box { size     3 3 3 }
            appearance Appearance {
                  texture ImageTexture {
                        url "t1.gif"
                  }
            }
      }
}

Transform {
      translation                  2 0 0
      children Shape {
            geometry Box { size     3 3 3 }
            appearance Appearance {
                  material Material {
                        diffuseColor 1 1 1
                        emissiveColor 0 .318 0
                  }
                  texture ImageTexture {
                        url "t1.gif"
                  }
            }
      }
}
```

Programa 4.8: Uso de imágenes como textura.

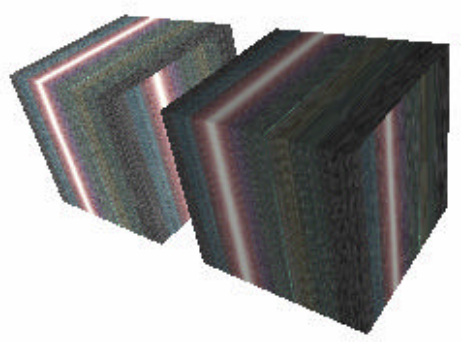


Figura 4.8: Uso de imágenes como textura.

El cubo de la izquierda solamente tiene el campo **texture** definido. Esto hace que el cubo no sea iluminado por ninguna de las fuentes de luz del entorno y queda con los colores originales de la textura. En cambio, el de la derecha también tiene definido el campo **appearance** con un color difuso. Esto hace que sí sea iluminado por las fuentes de luz del entorno.

El **Programa 4.9** muestra como queda aplicada la textura sobre cada una de las primitivas (**Figura 4.9**).

```
#VRML V2.0 utf8
DEF Textura Appearance {
            material Material { diffuseColor 1 1 1 }
            texture ImageTexture {
                  url "t1.gif"
            }
}
Transform {
      translation                  -2 2 0
      children Shape {
            geometry Box { size     3 3 3 }
            appearance USE Textura
      }
}
Transform {
      translation                  2 2 0
      children Shape {
            geometry Sphere { radius 1.5 }
            appearance USE Textura
      }
}
Transform {
      translation                  -2 -2 0
      children Shape {
            geometry Cone { height 3 bottomRadius 1.5 }
            appearance USE Textura
      }
}
Transform {
      translation                  2 -2 0
      children Shape {
            geometry Cylinder { height 3 radius 1.5 }
            appearance USE Textura
      }
}
```

Programa 4.9: Texturas definidas sobre las primitivas básicas.

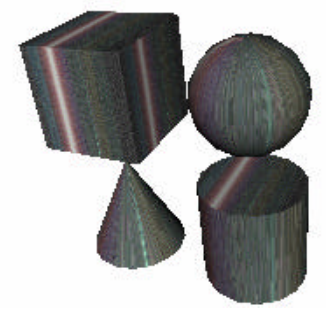


Figura 4.9: Texturas definidas sobre las primitivas básicas.

Para variar la medida y colocación de la textura sobre el objeto se puede utilizar el campo **textureTransform** de la apariencia. En este campo, hace falta utilizar el campo **TextureTransform** que permite escalar, rotar y trasladar la textura (**Programa 4.10** y **Figura 4.10**).

```
#VRML V2.0 utf8

Shape {
      geometry Box { size           3 3 3 }
      appearance Appearance {
            material Material {
                  diffuseColor      1 1 1
            }
            texture ImageTexture {
                  url "t1.gif"
            }
            textureTransform TextureTransform {
                  scale             2 2
                  rotation          1.5708 # PI/2
            }
      }
}
```

Programa 4.10: Transformación de una textura.

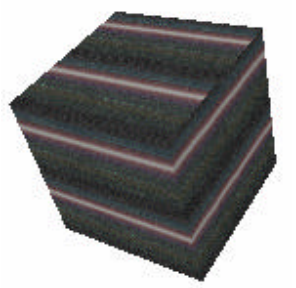


Figura 4.10: Transformación de una textura.

Hay que tener en cuenta que lo que estamos haciendo es escalar, rotar y trasladar las coordenadas de textura y no la textura en sí misma. Por lo tanto, un escalado de **[2 2]** no la amplía al doble, sino que la reduce a la mitad (y por lo tanto se repite), y una rotación de **p/2** no rota la textura **p/2**, si no **-p/2**, etc.

Para finalizar se muestran las definiciones de los formatos de imágenes y películas que pueden ser usadas en **VRML** **[Alchemy 96**, **Foley 96** y **Johnson 87]**:

- **JPEG** (**J**oin **P**hotographic **E**xpert **G**roup).- es un algoritmo diseñado para comprimir imágenes con **24** bits de profundidad o en escala de grises. **JPEG** es también el formato de archivo que utiliza este algoritmo para comprimir imágenes. **JPEG** sólo trata imágenes fijas, pero existe un estándar relacionado llamado **MPEG** para videos. El formato de archivos **JPEG** se abrevia frecuentemente **JPG** debido a que algunos sistemas operativos sólo aceptan tres letras de extensión.

- **PNG** (**P**ortable **N**etwork **G**raphics).- es un formato gráfico basado en un algoritmo de compresión sin pérdida para **bitmaps** (**M**apa de **B**its) no sujeto a patentes. Este formato fue desarrollado en buena parte para solventar las deficiencias del formato **GIF** y permite almacenar imágenes con una mayor profundidad de color y otros datos importantes.

- **GIF** (**G**raphics **I**nterchange **F**ormat).- es un formato de imagen utilizado ampliamente en la **World Wide Web**, tanto para imágenes como para animaciones. El formato fue creado por **CompuServe** en **1987** para dotar de un formato de imagen a color para sus áreas de descarga de archivos, sustituyendo su temprano formato **RLE** en blanco y negro. **GIF** llegó a ser muy popular porque podía usar el algoritmo de compresión **LZW** (**L**empel **Z**iv **W**elch) para realizar la compresión de la imagen, que era más eficiente que el algoritmo **Run-Lenght Encoding** (**RLE**) que usaban formatos como **PCX** y **MacPaint**. Por lo tanto, imágenes de gran tamaño podían ser descargadas en un periodo razonable de tiempo, incluso con módem muy lentos. **GIF** es un formato sin perdida de calidad, siempre que partamos de imágenes de **256** colores o menos. Una imagen de alta calidad, como una imagen de color verdadero (profundidad de color de **24** bits o superior) debería reducir literalmente el número de colores mostrados para adaptarla a este formato, y por lo tanto existiría una perdida de calidad.

- **MPEG** (**M**oving **P**icture **E**xpert **G**roup).- es el grupo encargado del desarrollo de normas de codificación para audio y vídeo, formado en el **C**omité **T**écnico para la **T**ecnología de la **I**nformación **ISO/IEC JTC 1**, de la **ISO**. Desde su primera reunión en **1988**, el **MPEG** ha crecido hasta incluir **350** miembros de distintas industrias y universidades. La designación oficial del **MPEG** es **ISO/IEC JTC1/SC29 WG11**. **MPEG** ha normalizado los siguientes formatos de compresión y normas auxiliares:

    - **MPEG-1**: estándar inicial de compresión de audio y vídeo. Usado después como la norma para **CD** de vídeo, incluye el popular formato de compresión de audio **Capa 3** (**MP3**).
    - **MPEG-2**: normas para audio y vídeo para difusión de calidad de **televisión**. Utilizado para servicios de **TV** por satélite como **Direct TV** (Cadena estadounidense de televisión vía satélite de difusión directa), señales de televisión digital por cable y (con ligeras modificaciones) para los discos de vídeo **DVD**.
    - **MPEG-3**: diseñado originalmente para **HDTV** (**T**elevisión de **A**lta **D**efinición), pero abandonado posteriormente en favor de **MPEG-2**.
    - **MPEG-4**: expande **MPEG-1** para soportar objetos audio/vídeo, contenido **3D**, codificación de baja velocidad binaria y soporte para gestión de derechos digitales (protección de copyright).
    - **MPEG-7**: sistema formal para la descripción de contenido **multimedia**.
    - **MPEG-21**: **MPEG** describe esta norma futura como un **marco multimedia**.

Capítulo 5: Prototipos

Cuando se modelan objetos para un entorno, nos podemos encontrar que necesitemos definir un tipo de objeto que será utilizado de formas diferentes en diferentes lugares del entorno **[Alarcón 00** y **VRML 97]**. Para poder definir un **objeto genérico** y poder utilizarlo con parámetros diferentes en diferentes lugares, **VRML 2.0** dispone del prototipo **PROTO**.

5.1 Definición de un prototipo

Hace falta tener claro que **PROTO** no es un nodo de **VRML 2.0**, sino un mecanismo para definir **nuevos nodos**. Lo que se hace al definir un **prototipo**, es decir, un tipo de objeto nuevo para poder utilizarlo de la misma forma en que utilizamos los nodos **Box**, **Sphere**, **Material**, **Appearance**, etc. En el **Programa 5.1** se tiene la definición de un prototipo de **Paraguas**. Se define el nuevo tipo de nodo **Paraguas**.

```
#VRML V2.0 utf8
PROTO Paraguas [
      field SFColor colorBaston           0.1 0.1 0
      field SFColor colorRopa             0   0.2 1
      ]
{
Group       {
      children [
            Shape {
                  geometry Cylinder {height 3 radius 0.1}
                  appearance Appearance {
                        material Material { diffuseColor IS colorBaston }
                        }
                  }
            Transform {
                  translation             0 1.75 0
                  children Shape {
                        geometry Cone {
                              height       0.5
                              bottomRadius 3
                              }
                        appearance Appearance {
                              material
                              Material { diffuseColor IS colorRopa }
                              }
                        }
                  }
            ]
      }
}

Paraguas { }      # Objeto paraguas con los valores por defecto
Transform {       # Objeto paraguas con una traslación positiva en X
      translation 8 0 0
      children Paraguas { colorBaston 0.2 0 0 colorRopa 0 0.2 0.1 }
      }
Transform {       # Objeto paraguas con una traslación negativa en X
      translation -8 0 0
      children Paraguas { colorBaston 0 0 0.3 colorRopa 0.4 0 0.1 }
      }
```

Programa 5.1: Definición de **prototipos**

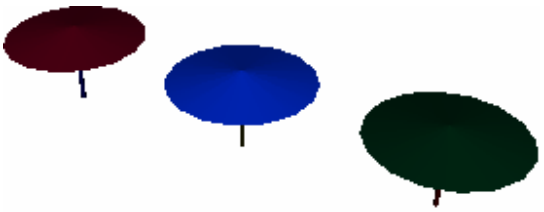


Figura 5.1: Definición de **prototipos**.

Se comienza definiendo el **prototipo** con la palabra **`PROTO`**. A continuación, se define el mundo que tenga el nuevo tipo de nodo que definimos, en este caso **`Paraguas`**. Entonces hace falta definir los parámetros que podremos variar de nuestro prototipo cada vez que necesitemos uno. En el **Programa 5.1** se pueden definir dos colores, se definen con el tipo **`SFColor`** (**S**ingle-valued **F**ield **Color**, campo de color univaluado).

A cada uno le damos el color por defecto, para que si alguien define un objeto de tipo **`Paraguas`** y no le da valores a los parámetros, **VRML** tenga un color para utilizar. Ahora ya podemos definir como serán las características generales de nuestro nuevo nodo **`Paraguas`**.

Para comenzar, utilizamos un nuevo nodo que es el nodo **`Group`**. Este nodo sirve para **agrupar** en un solo objeto varios subobjetos. Este nodo nos agrupará el objeto **bastón** y el objeto **sombrilla** de nuestro paraguas. Si no lo hiciéramos y definiéramos el objeto **bastón** y, a continuación, el objeto **sombrilla**, el prototipo estaría formado por un **bastón visible** y por una **sombrilla invisible**. En la definición de un prototipo se pueden poner tantos objetos como se quieran, pero siempre se tomará el primero para determinar el tipo de nodo que se está definiendo y para visualizarlo. El resto de los objetos pueden servir para muchas cosas, pero no serán visualizados.

Así pues, definimos un cilindro que nos hará de bastón. Como se puede observar, la geometría del cilindro es fija. En cambio, la apariencia es aquello que queremos variar. Por tanto, definimos una apariencia con un material y un color. La forma de conseguir que el color sea el que nosotros queremos, es utilizando la palabra **`IS`**. Esta palabra hace que podamos definir que el campo **`diffuseColor`** de nuestro objeto sea en realidad el campo **`colorBaston`** que hemos definido en los parámetros.

La utilización del comando **`PROTO`** permite crear nuevos nodos. De forma análoga se define la parte de la sombrilla que definimos mediante un cono trasladado hacia arriba para que enganche bien con el bastón.

Hasta aquí hemos definido el prototipo y, como tal, no es un objeto que exista. Lo que hacemos después es lo que se llama **instanciar un objeto a partir del prototipo**. Por el solo hecho de poner el nombre del nuevo nodo (**`Paraguas`**), ya estamos creando una instancia del prototipo. Como a continuación solamente ponemos unas llaves vacías, no estamos definiendo parámetros para este objeto en concreto. En este caso se utilizan los parámetros por defecto que hemos definido dentro del prototipo. También podemos ver que el objeto aparece en el espacio, en el lugar donde ha estado definido.

Las otras dos instancias se hacen trasladando el paraguas a un lado y al otro del primero y dando colores diferentes de los colores por defecto. Ha de quedar claro que al instanciar los tres paraguas obtenemos tres objetos diferenciados y que, si modificamos uno, los otros no se verían afectados.

El **Programa 5.2**, muestra la definición de un prototipo de **`Banca`**, para crear una mesa con sillas que se observa en la **Figura 5.2**.

```
#VRML V2.0 utf8

PROTO Banca [
      field SFColor colorpatas      1 1 0
      field SFColor colorBase       0 0 1
      field SFVec3f proporciones    1 1 1
      field SFVec3f posicion        0 0 0
      ]
{
PROTO Pata [
      field SFColor colorPata       1 1 0
      field SFVec3f posicion        0 0 0
      ]
      {
      Transform {
            translation IS posicion children Shape {
                  geometry Cylinder { radius 0.1 height 1}
                  appearance Appearance { material
                        Material { diffuseColor IS colorPata }
                        }
                  }
            }
      }
Transform {
      translation IS posicion
      scale IS proporciones
      children [
            Shape{ # Base
                  geometry Box { size 1 0.2 1 }
                  appearance Appearance { material Material {
                              diffuseColor IS colorBase
                              }
                        }
                  }
            Pata { posicion 0.4 -0.6 0.4
                  colorPata IS colorpatas
                  }
            Pata { posicion -0.4 -0.6 0.4
                  colorPata IS colorpatas
                  }
            Pata { posicion -0.4 -0.6 -0.4
                  colorPata IS colorpatas
                  }
            Pata { posicion 0.4 -0.6 -0.4
                  colorPata IS  colorpatas
                  }
            ]
      }
}

Banca { }
Banca { colorpatas 1 0.5 0 colorBase 0.5 0 1 posicion -2 0 0 }
Banca { colorpatas 0.5 1 0 colorBase 1 0.5 0 posicion  2 0 0 }
Banca { colorpatas 1 0 0.8 colorBase 1 0.5 0 posicion  0 2 0  proporciones 3 1 1 }
```

Programa 5.2: Prototipo de **Banca**.

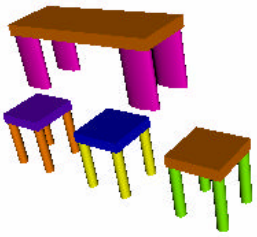


Figura 5.2: Resultados del prototipo de **Banca**.

En este ejemplo vemos un prototipo de **Banca** que se apoya en un prototipo de **pata** para definir sus cuatro patas. Esta **banca** ha sido definida como un nuevo tipo de nodo.

Hace falta notar como los nombres de los campos no se solapan entre el **PROTO** padre e hijo. Es decir que, por ejemplo, podemos tener como campo el nombre **posición** tanto en uno como en el otro y se sabe perfectamente a cual se está haciendo referencia en cada momento, debido al ámbito donde se encuentra.

5.2 DEF y USE

Cuando lo se quiere es definir un objeto que se utiliza idéntico en muchos lugares, entonces podemos utilizar **DEF** y **USE**. Hace falta no confundir estos dos comandos con **PROTO**.

DEF y **USE** no definen un objeto genérico que se instancia como objeto particular al llamarlo con unos ciertos parámetros. Lo que se hace en realidad es dar **nombre** a un cierto **nodo** que después utilizaremos en muchos lugares de la misma forma.

```
#VRML V2.0 utf8
DEF RojoMio Appearance {
      material Material {
            diffuseColor      0.8 0.2 0.4
            }
      }
Transform {
      translation             -2 0 0
      children Shape {
            geometry Box { size 3 3 1 }
            appearance USE RojoMio
            }
      }
Transform {
      translation             2 0 0
      children Shape {
            geometry Sphere { radius 2 }
            appearance USE RojoMio
            }
      }
```

Programa 5.3: Utilización de **DEF** y **USE**.

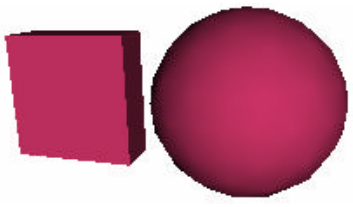


Figura 5.3: Utilización de **DEF** y **USE**.

Aquí vemos como hemos definido (**DEF**) una apariencia que después utilizamos (**USE**) en dos objetos diferentes. Lo que hemos hecho es sencillamente darle el nombre **RojoMío** a un trozo de código para después ahorramos escribirlo muchas veces. Si tenemos que hacer un cambio de color de la apariencia, se hace el cambio en un solo lugar (en **DEF**), se tendrá hecho en todos los lugares donde lo hemos utilizado (**USE**).

Por ejemplo, si queremos que el color **RojoMio** sea más rojo; sólo nos hace falta cambiar su definición en **DEF** (**Programa 5.4** y **Figura 5.4**).

```
#VRML V2.0 utf8
DEF RojoMio Appearance {
      material Material {
            diffuseColor      1 0.2 0
            }
      }
Transform {
      translation             -2 0 0
      children Shape {
            geometry Box { size 3 3 3 }
            appearance USE RojoMio
            }
      }

Transform {
      translation             2 0 0
      children Shape {
            geometry Sphere { radius 2 }
            appearance USE RojoMio
            }
      }
```

Programa 5.4: Utilización de **DEF** y **USE**.

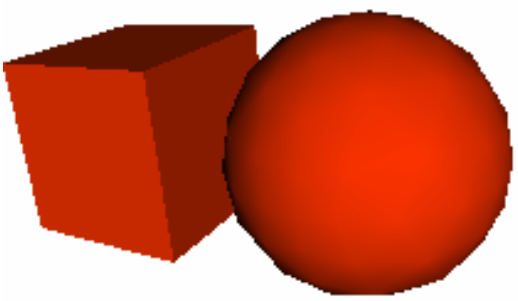


Figura 5.4: Utilización de **DEF** y **USE**.

Capítulo 6: Anchors, Billboarding y Colisiones

6.1 Anchors (Enlaces)

Así como en **HTML** (**H**yper**T**ext **M**arkup **L**anguage) se pueden definir **enlaces** (**link**) a otros documentos, en **VRML** también se puede hacer [Alarcón 00 y VRML 97]. Esto se consigue mediante el nodo **Anchor**. Este nodo permite englobar a un objeto o conjunto de objetos sobre un mismo **enlace**, de forma que cuando el usuario señale uno de los objetos con el cursor y pulse el botón del ratón, se genera el enlace y se abre el documento asociado, sea **HTML**, **VRML** o cualquier otra posibilidad.

Cada navegador tiene su forma para indicar que un objeto es activo como enlace pero, en general, se acostumbra a cambiar la forma del cursor o a aumentar la intensidad de color del objeto cuando se pasa el cursor por encima del mismo. En el **Programa 6.1** se observa la definición de un enlace a una página de **Internet** (Figura 6.1).

```
#VRML V2.0 utf8
Anchor {
      children Shape {
            geometry Sphere { radius 2 }
            appearance Appearance {
                  material Material {diffuseColor 0.8 0.5 0 }
                  }
            }
      url "http://www.geocities.com/daraujo14"
      }
```

Programa 6.1: Enlace con un nodo **Anchor**.

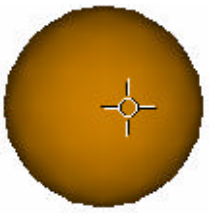


Figura 6.1: Enlace con un nodo **Anchor**.

Como se puede ver, solamente se han de incluir los objetos que queremos que sean enlaces como a hijos del nodo **Anchor** y ya tenemos el enlace hecho.

Hay un campo que permite poner toda clase de parámetros referidos al enlace. Por ejemplo, a qué **frame** ha de ir a parar el documento enlazado (**Programa 6.2** y **Figura 6.2**).

```
#VRML V2.0 utf8
Anchor {
      children Shape {
            geometry Sphere { radius 2 }
            appearance Appearance {
                  material Material {diffuseColor 0.8 0.5 0 }
                  }
            }
      url "http://www.geocities.com/daraujo14"
      parameter [ "target=nouframe" ]
      }
```

Programa 6.2: Enlace con **target**.

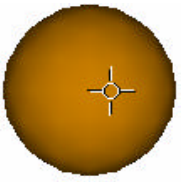


Figura 6.2: Enlace con **target**.

6.2 Viewpoints como Anchors

Los nombres que damos a los **puntos de vista** pueden servir como puntos de entrada al mundo desde otro mundo o desde un documento **HTML**, de la misma forma que podemos definir **Anchors** locales en **HTML**. La forma de hacerlo es añadiendo el nombre del punto de vista en la **URL** (dirección) del mundo poniéndole un símbolo # delante.

Por ejemplo para mostrar el **punto de vista** de **Telefoto** de un archivo **VRML** **p_2_05.wrl**, se tiene el **Programa 6.3**.

```
#VRML V2.0 utf8
Anchor {
      children Shape {
            geometry Sphere { radius 2 }
            appearance Appearance {
                  material Material {diffuseColor 0.8 0.5 0 }
                  }
            }
      url "../p_2_05.wrl#Telefoto"
      }
```

Programa 6.3: Enlace a un **Viewpoint**.

6.3 Insertar mundos (Inline)

Hay veces que nos interesa incluir algún mundo ya creado dentro del que estamos construyendo. Esto puede ser para reutilizar objetos o porque es un mundo que no hemos creado nosotros. El nodo **Inline** nos permite hacer esto (**Programa 6.4** y **Figura 6.3**).

```
#VRML V2.0 utf8
Shape {
      geometry Sphere { radius 0.5 }
      appearance Appearance {
            material Material {diffuseColor 0.8 0.5 0 }
            }
      }
Inline      { url "p_5_01.wrl" }
```

Programa 6.4: Uso de **Inline**.

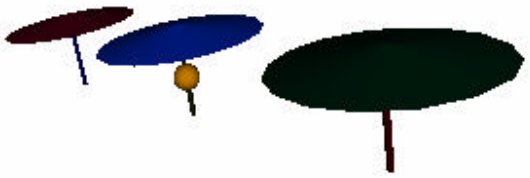


Figura 6.3: Uso de **Inline**.

Lo que se ha hecho es incluir en este nuevo mundo el código del ejemplo de la definición de paraguas. Por lo tanto, aparecen los objetos definidos en ese mundo más la esfera definida en el primero.

Se pueden poner más de una **URL**, separadas por comas y agrupadas dentro de corchetes **[]**. Con esto, definimos una lista en orden decreciente de preferencia. Por ejemplo, si el navegador no puede acceder al archivo de la primera **URL**, entonces intenta acceder a la segunda, etc.

6.4 Billboarding

Esta técnica es ampliamente usada en gráficos **3D** de tiempo real para representar objetos que tienen una geometría demasiado compleja como para ser modelados con polígonos. Lo que se hace es texturar un polígono con la textura del objeto en cuestión y entonces se le da la propiedad de **Billboard**. Así se logra que el polígono siempre esté encarado hacia el observador. Con esto se consigue que nunca se descubra el truco, ya que el usuario no puede ver nunca el polígono de lado. En **VRML 2.0** esto se consigue gracias al nodo **Billboard** (**Programa 6.5** y **Figura 6.4**).

```
#VRML V2.0 utf8
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 "flores.gif"
                        }
                  }
            }
      }
```

Programa 6.5: Uso del **Billboard**.

Figura 6.4: Uso del **Billboard**.

Lo que definimos es un polígono con las proporciones que tiene nuestra textura de la flor (para que no se deforme), le aplicamos la textura de flor con transparencia y lo definimos como hijo del nodo **Billboard**. El otro dato que hace falta definir es el eje de rotación respecto el cual ha de pivotar nuestro polígono.

Como el eje que hemos definido es el **eje Y**, sólo nos seguirá si nos movemos en un plano perpendicular a este eje. Si nos movemos hacia arriba o hacia abajo, podemos perder el efecto deseado.

6.5 Colisiones

La detección de **colisiones** puede ser muy importante en algunos casos; **VRML 2.0** permite una **detección parcial**. Parcial en el sentido de que solamente da mecanismos para detectar colisiones entre el observador y los objetos, pero no permite la detección de colisiones entre objetos.

El mecanismo para activar y desactivar la detección de colisiones funciona a partir de un nodo de agrupamiento, el nodo **Collision**. Éste se utiliza de forma que agrupa una lista de objetos, los cuales no se quiere que el observador atraviese (**Programa 6.6** y **Figura 6.5**).

```
#VRML V2.0 utf8
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 {
                        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 } }
                        }
                  }
            ]
      }
```

Programa 6.6: Activación de las **colisiones**

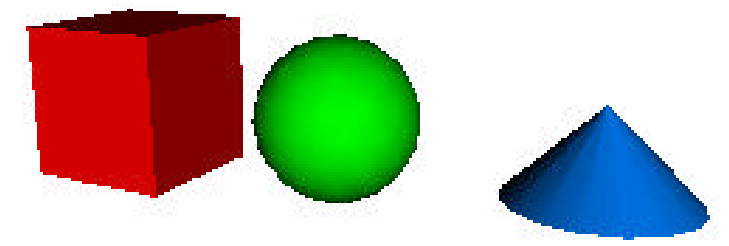


Figura 6.5: Activación de las **colisiones**

En este ejemplo, el observador puede atravesar el cubo pero no puede atravesar la esfera y el cono. Como se puede ver, ponemos el cubo dentro de un grupo de colisión con la propiedad de colisión **FALSE**. Esto lo hacemos porque la especificación de **VRML 2.0** dice que los navegadores, por defecto, deberán de detectar colisiones con todos aquellos objetos en los que no se haya definido el comportamiento explícitamente.

6.6 Eventos

Un **evento** es un mensaje que puede ser enviado por un objeto y recibido por otro. Puede haber eventos indicando que un nodo ha cambiado de posición, que ha pasado de estar inactivo a activo, que ha pasado una fracción de tiempo, etc.

Los navegadores tienen un **reloj interno** que va marcando en tiempo real del paso del tiempo. Los eventos se van produciendo y recibiendo en cada intervalo de tiempo del reloj (en teoría de forma continua) pero, en la práctica, podemos asumir que los navegadores marcan el paso del tiempo de cada **frame** (cuadro) generado gráficamente. Así pues, cada evento tiene una especie de sello de tiempo del momento en que ha sido generado **(emitido, enviado)**.

Evidentemente, un evento es **emitido** por un **eventOut** y ha de **recibir** un **eventIn**. Pero también puede pasar que un evento sea emitido o recibido por un **exposedField** que son campos de **entrada** y **salida**.

Tanto el evento que envía como el que recibe han de ser del mismo tipo. Por eso, un evento enviado por un **eventOut** de tipo **SFInt32** (entero de 32 bits univaluado), sólo puede ser recibido por un **eventIn** (o **exposedField**) de tipo **SFInt32**.

Puede pasar que un objeto reciba un evento que fuerce a enviar una respuesta. En este caso se genera lo que se conoce por **cascada de eventos**. En una cascada de eventos, todos los eventos implicados tienen el mismo sello de tiempo.

Pero ¿qué pasa si se genera un **loop** (lazo ó bucle)? Podría pasar que nunca se acabase de enviar eventos y no avanzase **Fan Out**. Este problema no existe en **VRML 2.0** porque no se permite que un nodo reciba en un **eventIn** un evento proveniente del mismo lugar con el mismo sello de tiempo. Si se detecta un **loop**, entonces el **VRML 2.0** se encarga de pasar el evento de tiempo (T_i) al tiempo (T_{i+1}).

Es permitido que un **eventOut** pueda ser encaminado a múltiples **eventIn**. Esto es conocido como **Fan Out**. En cambio, no es legal encaminar múltiples **eventOut** a un sólo **eventIn**. Esto es conocido como **Fan In**. Si se produce un **Fan In**, se obtienen resultados indefinidos.

6.6.1 Routing de valores (comando route)

Pero ¿cómo podemos decir que un cierto **eventOut** ha de estar enlazado a un cierto **eventIn**? Para esto, **VRML 2.0** provee un mecanismo llamado **ROUTE**, el cual se consigue mediante el comando **ROUTE** (hace falta notar que no es un nodo, sino sólo un comando que sirve como un mecanismo).

Por ejemplo, supongamos que, de alguna manera, el color de un material cambia y queremos que el evento (tipo **SFColor**) lo reciba con una luz direccional, modificando así el color de la luz. Entonces, los nodos y la ruta en cuestión quedan como se muestra en el **Programa 6.7** y en la **Figura 6.6**.

```
#VRML V2.0 utf8

Shape {
      geometry Sphere { radius 2 }
      appearance Appearance {
            material DEF MaterialEsfera Material { diffuseColor 0 0 1 }
            }
      }

DEF Llum DirectionalLight {
      color       1 1 1
      direction   0.5 0.5 0
      intensity   1
      }

ROUTE MaterialEsfera.diffuseColor TO Llum.color
```

Programa 6.7: Definición de un **ROUTE** para capturar un evento de salida.

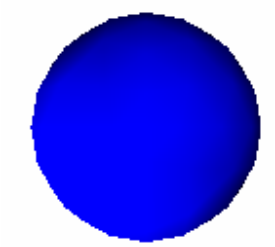


Figura 6.6: Definición de un **ROUTE** para capturar un evento de salida.

Como vemos, tenemos que dar un nombre a los nodos que queremos enlazar para poderlos referenciar. Una vez definidos los nombres (con **DEF**), podemos establecer enlaces con el **ROUTE**.

El **ROUTE** nos dice que los eventos emitidos por el **exposedField diffuseColor** del **Material MaterialEsfera** han de ser encaminados al (**TO**) **exposedField color** del **DirectionalLight Llum**.

Como se puede ver, la forma de decir cual **eventIn**, **eventOut** o **exposedField** vamos a tratar de un cierto nodo es lo que nos interesa. Esto se hace mediante un punto (**.**) que une el nombre del nodo al del campo seleccionado.

Capítulo 7: Sensores e Interpoladores

7.1 Sensor de tiempo

VRML 2.0 dispone de una herramienta muy potente: el **Sensor de Tiempo** (`TimeSensor`) **[Alarcón 00** y **VRML 97]**. Éste tiene un **reloj** que podemos utilizar a nuestro gusto para poder aprovechar el paso del tiempo como motor para mover los objetos de lugar, cambiar colores de las cosas, variar orientaciones, etc.

Imaginemos que queremos que un objeto gire alrededor de uno de sus ejes al ir pasando el tiempo. Tenemos un reloj que en cada *TICTAC* podemos definir que nos ayude a modificar alguna variable de nuestro mundo con una cierta frecuencia. Si de alguna forma pudiéramos redirigir el cambio de tiempo que hay en el reloj para variar la rotación del objeto, ya lo tendríamos solucionado.

Un `TimeSensor` se basa en el reloj de tiempo real del sistema, midiendo el tiempo actual, en segundos transcurridos a partir de las **00:00 horas** del **1ro** de **enero de 1970**.

En el **Programa 7.1** se hace un intervalo de **5 segundos**.

```
#VRML V2.0 utf8

DEF Finito TimeSensor{
      startTime         0
      cycleInterval     5
      }
```

Programa 7.1: Intervalo de **5 segundos**

Este `TimeSensor` se pone en marcha al cargarse el entorno en el visualizador y se para tras transcurrir **5 segundos**. Un `TimeSensor` constantemente da como **evento de salida** (`eventOut`) la fracción de ciclo que lleva transcurrido.

Esto lo hace mediante el `eventOut SFFloat fraction_changed`, que tiene un valor entre `0.0` y `1.0`. La duración del ciclo se define en segundos en el `exposedField SFTime cycleInternal`, que en este caso hemos puesto en **5 segundos**. Es decir, el `TimeSensor` tarda **5 segundos** en pasar de `0.0` a `1.0` el valor de `fraction_changed`. Cuando hayan transcurrido los **5 segundos** y, por lo tanto, ha llegado a `1.0` el valor de `fraction_changed`, el `TimeSensor` dejará de modificar el `fraction_changed` y se parará.

Cuando el **Programa 7.2** se pone en marcha, va haciendo ciclos de **5 segundos**.

```
#VRML V2.0 utf8

DEF Infinito TimeSensor{
      startTime         0
      cycleInterval     5
      loop TRUE
      }
```

Programa 7.2: Intervalos de **5 segundos**.

Ahora, al haber transcurrido los **5 segundos** y haber llegado a `1.0`, el valor de `fraction_changed`, como le hemos dicho que haga ciclos con el `loop TRUE`, volverá a comenzar otro ciclo. Esto lo irá haciendo sin pararse hasta que carguemos otro mundo **VRML** en el visualizador **VRML**.

Para poder aprovechar el paso del tiempo que nos va dando el `TimeSensor`, hace falta utilizar los interpoladores.

7.2 Interpoladores

La **interpolación lineal** (que es la que utiliza **VRML 2.0**) es un concepto matemático que permite definir dos puntos (en cualquier dimensión) y calcular un punto intermedio sobre la recta que los une. Para ello debemos concretar qué tanto por ciento del recorrido entre los dos puntos queremos hacer.

En **VRML 2.0** hay seis tipos de **interpoladores**:

1. `ColorInterpolator` interpola colores.
2. `CoordinateInterpolator` interpola coordenadas de vértices.
3. `NormalInterpolator` interpola normales a superficies.
4. `OrientationInterpolator` interpola ángulos de rotación.
5. `PositionInterpolator` interpola posiciones de objetos.
6. `ScalarInterpolator` interpola valores cualesquiera (escalares, es decir, univaluados).

Comenzaremos por el **interpolador de color**. Lo que queremos hacer es que un objeto vaya cambiando de color de forma gradual, comenzando por el rojo y acabando por el verde. Sabemos que, según la forma de definir los colores en **RGB** (valores entre `0` y `1`), el rojo es el (`1 0 0`) y el verde es el (`0 1 0`).

Ahora tramos el cambio de color y por lo tanto definimos nuestro **interpolador de color** de la siguiente forma:

```
DEF Arco ColorInterpolator {
      key      [     0,   0.3,   0.6,     1]
      keyValue [ 1 0 0, 0 1 0, 0 0 1, 1 0 0]
      }
```

Con esta definición, estamos diciendo que queremos que, cuando el interpolador comience (`0%` del trayecto, `key = 0`), nos dé el color rojo `keyValue = (1 0 0)`. Cuando esté a `key = 0.3` (aproximadamente un tercio del trayecto) nos dé el color verde `keyValue = (0 1 0)`. Cuando esté a `key = 0.6` (aproximadamente dos tercios) nos dará color azul `keyValue = (0 0 1)`. Y cuando acabe (`100%` del trayecto, `key = 1`) dará el color rojo de nuevo, `keyValue = (1 0 0)`.

De esta forma, si tenemos definido un material que aplicamos a un objeto, podemos hacer que el interpolador vaya modificando el `diffuseColor` y por lo tanto, nos vaya cambiando el color del objeto al cual lo aplicamos. De alguna forma hemos de conectar el cambio de color del interpolador al color del material.

Pero, ¿qué es lo que hace que el interpolador vaya pasando del valor inicial al final poco a poco? Pues el paso del **tiempo** del `TimeSensor` que hemos visto anteriormente. Aquí también hemos de conectar el cambio de tiempo del `TimeSensor` al cambio de tanto por ciento del interpolador (`key`).

La herramienta necesaria para conectar es la **ruta**. Veamos como funciona todo nuestro ejemplo del objeto (un cubo) que va cambiando de color (**Programa 7.3** y **Figura 7.1**).

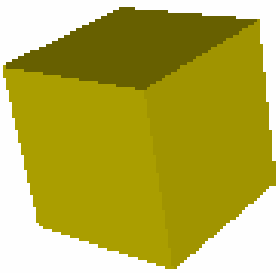


Figura 7.1: Cubo que **cambia de color**.

```
#VRML V2.0 utf8
DEF motorColor TimeSensor{
      loop              TRUE
      cycleInterval     5
      }
DEF Arco ColorInterpolator {
      key               [     0,   0.3,   0.6,     1]
      keyValue          [ 1 0 0, 0 1 0, 0 0 1, 1 0 0]
      }
DEF CubColorCambiante Shape {
      geometry Box { size 2 2 2 }
      appearance Appearance { material DEF ColorCambiante Material {} }
      }
ROUTE motorColor.fraction_changed TO Arco.set_fraction
ROUTE Arco.value_changed TO ColorCambiante.diffuseColor
```

Programa 7.3: Cubo que **cambia de color**.

Analicemos el programa:

- En primer lugar, definimos un **TimeSensor** que llamamos **motorColor**. Este **TimeSensor** se pondrá en marcha al cargar el entorno e irá haciendo ciclos de **5** segundos sin pararse nunca.
- A continuación, definimos un **ColorInterpolator** de tres tramos (rojo-verde, verde-azul, azul-rojo) de la misma longitud, aproximadamente un tercio cada uno.
- En tercer lugar, definimos nuestro objeto, el **CubColorCambiante** que, como su nombre indica, es un cubo que va cambiando de color. Este objeto se define como una geometría que es una caja (**Box**) que forma un cubo y por una apariencia con un material. El **material** tiene asignado un nombre, **ColorCambiante** pero no tiene ningún parámetro definido en **Material**. Esto es porque el campo **diffuseColor** será tratado como un evento de entrada (**eventIn**).
- Finalmente encontramos las rutas:
  - La primera define que el paso del **TimeSensor** vaya haciendo avanzar el trayecto recorrido por el interpolador de color. Esto se consigue enlazando el evento de salida **fraction_changed** del **TimeSensor motorColor** con el evento de entrada **set_fraction** del **Arco ColorInterpolator**. Con este enlace, cada *tictac* del **TimeSensor** hace incrementar un poquito el avance del **ColorInterpolator**.
  - La segunda enlaza el nuevo color que da el **ColorInterpolator** después de haber avanzado (debido a la acción del **TimeSensor**) con el color del objeto. Esto se consigue enlazando el **eventOut value_changed** del **ColorInterpolator** con el **eventIn diffuseColor** del material **ColorCambiante** del cubo que hem os definido.

De la misma forma que hemos enlazado un **TimeSensor** a un **ColorInterpolator**, lo podemos hacer a cualquier otro interpolador.

7.3 Sensor de Proximidad

El **sensor de proximidad** es una herramienta muy interesante para poder hacer un seguimiento de los movimientos del observador en una cierta zona del entorno. El nodo **ProximitySensor** nos permite definir una zona en forma de caja centrada en una cierta posición del espacio, la cual detecta si el observador está dentro o fuera, en qué instante entra o sale, etc.

A continuación, damos un ejemplo donde sale un **TimeSensor** que se pone en marcha o se apaga, en función de si el observador está dentro o fuera de una zona cúbica del espacio. Para ver como se pone en marcha y se apaga el **TimeSensor**.

En el **Programa 7.4** tenemos un **sensor de proximidad** que pone en marcha y apaga el cambio de color del cubo (**Figura 7.2**).

```
#VRML V2.0 utf8

DEF motorColor TimeSensor{
      loop              TRUE
      cycleInterval     5
      }

DEF Arco ColorInterpolator {
      key               [     0,   0.3,   0.6,     1]
      keyValue          [ 1 0 0, 0 1 0, 0 0 1, 1 0 0]
      }

DEF CuboColorCambiante Shape {
      geometry Box { size 2 2 2 }
      appearance Appearance { material DEF ColorCambiante Material {} }
      }

DEF ZonaActiva ProximitySensor {
      center            0 0 0
      size              8 8 8
      }

ROUTE motorColor.fraction_changed TO Arco.set_fraction
ROUTE Arco.value_changed TO ColorCambiante.diffuseColor
ROUTE ZonaActiva.isActive TO motorColor.enabled
```

Programa 7.4: Sensor de **proximidad**.

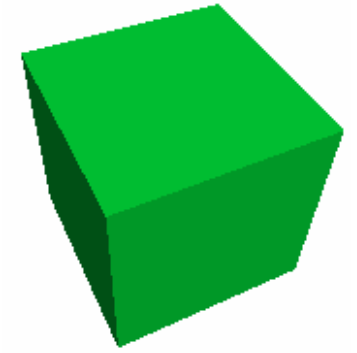


Figura 7.2: Sensor de **proximidad**.

Los elementos nuevos son el **sensor de proximidad** **ZonaActiva**, que define una zona de **8x8x8** unidades centrada en el origen de coordenadas **(0,0,0)** y el último **ROUTE**, que conecta el **eventOut isActive** del sensor de proximidad con el **exposedField enable** del sensor de tiempo.

Este conjunto detecta si el observador está fuera o dentro de la zona comprendida entre los planos **X=4** y **X=-4**, **Y=4** e **Y=-4** y **Z=4** y **Z=-4**. Si el observador está fuera, el **sensor de proximidad** emite un evento de **isActive = FALSE** y, por lo tanto, el **ROUTE** lo canaliza a desactivar el sensor de tiempo a través del **exposedField enable**. Si el observador está dentro, el sensor de proximidad emite un evento de **isActive = TRUE** y, por lo tanto, el **ROUTE** lo canaliza y activa el sensor de tiempo.

De esta forma, el cambio de color queda congelado mientas el observador está fuera de la zona y vuelve a variar cuando el observador entra.

7.4 Sensor de tacto

El nodo **TouchSensor** permite detectar cuándo el observador apunta a un objeto con el cursor y cuándo pulsamos el ratón. Por lo tanto, permite definir **botones 3D**.

Para ver como funciona, modificaremos el **Programa 7.5** y ahora el cambio de color del cubo comenzará cuando se pulse sobre el cilindro verde que hay en primer plano (**Figura 7.3**).

```
#VRML V2.0 utf8
DEF motorColor TimeSensor {
      loop TRUE
      cycleInterval     5
      enabled           FALSE
      }

DEF Arco ColorInterpolator {
      key               [     0,   0.3,   0.6,     1]
       keyValue         [ 1 0 0, 0 1 0, 0 0 1, 1 0 0]
      }

DEF CuboColorCambiante Shape {
      geometry Box { size 2 2 2 }
      appearance Appearance { material DEF ColorCambiante Material {} }
      }

DEF BotonVerde Transform {
      translation       0 -1 7
      children [
            Shape {
                  geometry Cylinder { height 0.1 radius 1 }
                  appearance Appearance {
                        material Material {diffuseColor 0 1 0 }
                        }
                  }
            DEF SensorBotonVerde TouchSensor { }
            ]
      }

ROUTE motorColor.fraction_changed TO Arco.set_fraction
ROUTE Arco.value_changed  TO ColorCambiante.diffuseColor
ROUTE SensorBotonVerde.isActive TO motorColor.enabled
```

Programa 7.5: Sensor de **Tacto**.

Figura 7.3: Sensor de **Tacto**.

Mientras se esté pulsando el botón verde, el **sensor de tacto** va emitiendo eventos de que está activo y se encaminan hacia el sensor de tiempo que activa el cambio de color. Al dejar de apretar, el sensor de tacto envía eventos de inactividad y detiene el sensor de tiempo, parando así el cambio de color.

Es curioso notar que el **sensor de tacto** ha de ser agrupado conjuntamente con los objetos que servirán de botón dentro de algún nodo de agrupación, para que **VRML** sepa qué objetos estarán activos al ser pulsados por el usuario. Por eso hemos definido el sensor de tacto **`SensorBotonVerde`** como hermano del cilindro que define la geometría del botón dentro del nodo agrupador **`BotonVerde Transform`**.

7.5 Sensor de visibilidad

Este sensor sólo sirve para cuestiones de **optimización**. Por ejemplo, si tenemos muchos elementos en movimiento en nuestro entorno, el rendimiento de navegación puede ser bastante bajo. Pero si hacemos que los objetos sólo se muevan cuando entran en nuestro campo de visión, mejoraremos el rendimiento porque sólo se moverá aquello que realmente es necesario.

Esto se puede hacer definiendo una zona en forma de caja alrededor de los objetos. El **sensor de visibilidad** **`VisibilitySensor`** nos permite definir esta caja invisible, centrada en un cierto punto. Cuando esta caja entra en nuestro rango de visión, entonces el sensor de visibilidad se activa y emite un evento.

En el **Programa 7.6** se muestra un sensor de visibilidad, para ver la esfera el observador tiene que encontrarse dentro del cubo donde es visible.

```
#VRML V2.0 utf8

Transform {
      translation       -5 8 3
      children Shape {
            geometry Sphere { radius 2 }
            appearance Appearance {  }
            }
      }

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

Programa 7.6: Sensor de **Visibilidad**.

Este ejemplo nos muestra como definir la **caja de visibilidad** alrededor de un objeto. Como la esfera está centrada en el punto **`(-5 8 3)`** y tiene un radio de **`2`** unidades, entonces hemos de centrar la caja de visibilidad en el mismo punto y darle unas medidas de **`4x4x4`** para que englobe toda la esfera.

También algunos nodos tienen un parámetro de visibilidad, como el caso de definición de **neblina**, en el cual podemos definir el rango de visibilidad de un objeto. En el **Programa 7.7** se muestra como emplear el nodo **`Fog`** (neblina), para bloquear la visibilidad entre un objeto y el observador (**Figura 7.4**).

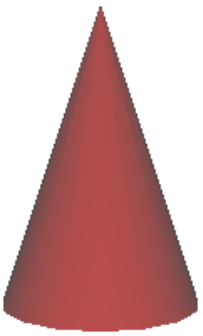


Figura 7.4: **Neblina**.

```
#VRML V2.0 utf8

Group {
      children [
            Fog   {
                  color 0 0 0
                  fogType "EXPONENTIAL"
                  visibilityRange 25
                  },
            DEF CenterPiece Transform {
                  children [
                        Shape {
                              appearance Appearance {
                                    material Material {
                                          diffuseColor 1 0 0
                                          }
                                    }
                              geometry Cone {
                                    bottomRadius 1
                                    height       3
                                    }
                              }
                        ]
                  }
            ]
      }
```

Programa 7.7: **Neblina**.

7.6 Sensor de movimiento

Estos sensores nos permiten incidir sobre las traslaciones y rotaciones de objetos directa e individualmente, sin tener que modificar toda la escena con el navegador.

Hay tres tipos de **sensores de movimiento** (*drag sensors*):

- **PlaneSensor**: permite modificar la **posición** del objeto, como si se moviera en un **plano**.
- **CylinderSensor**: permite modificar la **orientación** del objeto, como si **rotara** alrededor de un **eje**.
- **SphereSensor**: permite modificar la **orientación** de un objeto, como si **rotara** respecto a su **centro**.

Nodo PlaneSensor

Este sensor permite mapear los **movimientos 2D** del cursor de pantalla en **movimientos 2D** sobre el **plano XY** del sistema de coordenadas del sensor.

El sensor se activa cuando se pulsa sobre el objeto activo y, sin dejar el botón del ratón, se mueve el cursor. Entonces, los movimientos del cursor se pueden reflejar sobre el objeto mediante un **ROUTE**.

Veamos un ejemplo donde un cubo verde es el objeto activo y tenemos un cubo naranja de referencia inactivo (**Programa 7.8** y **Figura 7.5**).

Figura 7.5: Sensor de **movimiento**.

```
#VRML V2.0 utf8

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 Cubo Transform {          # Cubo que se mueve
                  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 Cubo.translation
```

Programa 7.8: Sensor de **movimiento**.

Este sensor se define como hermano de los objetos que han de ser activos.

En el **Programa 7.9** se observa un ejemplo, en donde un cubo es el activo y el que se mueve es el otro cubo (**Figura 7.6**).

Figura 7.6: Mapeo de **movimientos 2D**.

```
#VRML V2.0 utf8

Group {
      children [
            Transform {                  # Cubo activo
                  translation -3 0 0
                  children Shape {
                        geometry Box { size 2 2 2 }
                        appearance Appearance {
                              material Material { diffuseColor 0.7 0.3 0 }
                              }
                        }
                  }
            DEF PS PlaneSensor { }
            ]
      }

DEF Cubo Transform {                     # Cubo que se mueve
      children Shape {
            geometry Box { size 2 2 2 }
            appearance Appearance {
                  material Material { diffuseColor 0 0.7 0.3 }
                  }
            }
      }

ROUTE PS.translation_changed TO Cubo.translation
```

Programa 7.9: Mapeo de **movimientos 2D**.

Como se puede observar, el mapeo hecho con el **ROUTE** no ha variado, pero hemos puesto el sensor de movimiento en el plano, agrupado con el otro cubo (el naranja).

Nodo CylinderSensor

Este sensor permite mapear los **movimientos 2D** del cursor de pantalla en **rotaciones** alrededor del **eje Y** del sistema de coordenadas del sensor. El sensor se activa cuando se pulsa sobre el objeto activo y, sin dejar el botón del ratón, se mueve el cursor. Entonces los movimientos del cursor se pueden reflejar sobre el objeto mediante un **ROUTE**.

Veamos el ejemplo del cubo verde modificado (**Programa 7.10** y **Figura 7.7**).

Figura 7.7: Sensor cilíndrico.

```
#VRML V2.0 utf8

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 [                         # Cubo que se mueve
            DEF Cubo 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 Cubo.rotation
```

Programa 7.11: Sensor esférico.

Observe como este sensor se define de forma parecida al sensor **`CylinderSensor`**, es decir, hace falta definirlo como hermano de los objetos que han de ser activos.

7.7 Animación con sensores

Los sensores se emplean principalmente para **animar** algunos objetos dentro de un mundo virtual. En el **Programa 7.12** se observa el uso de los sensores de tiempo y de coordenadas para simular el movimiento de una lombriz (**Figura 7.9**).

Figura 7.9: **Animación** con sensores.

```
#VRML V2.0 utf8

Transform {
      translation                  0.0 0.3 0.0
      children Shape {
            appearance Appearance {
                  material Material { diffuseColor 0.0 1.0 0.2 }
                  }
            geometry DEF Cuerpo Extrusion {
                  creaseAngle       1.57
                  crossSection [    # Circular
                                    1.00   0.00,   0.92 -0.38,
                                    0.71  -0.71,   0.38 -0.92,
                                    0.00  -1.00,  -0.38 -0.92,
                                    -0.71 -0.71,  -0.92 -0.38,
                                    -1.00 -0.00,  -0.92  0.38,
                                    -0.71  0.71,  -0.38  0.92,
                                    0.00   1.00,   0.38  0.92,
                                    0.71   0.71,   0.92  0.38,
                                    1.00   0.00
                              ]
                  spine       [     # Senoide
                                    -4.100 0.0  0.000,  -4.000 0.0  0.000,
                                    -3.529 0.0  0.674,  -3.059 0.0  0.996,
                                    -2.588 0.0  0.798,  -2.118 0.0  0.184,
                                    -1.647 0.0 -0.526,  -1.176 0.0 -0.962,
                                    -0.706 0.0 -0.895,  -0.235 0.0 -0.361,
                                    0.235 0.0   0.361,   0.706 0.0  0.895,
                                    1.176 0.0   0.962,   1.647 0.0  0.526,
                                    2.118 0.0  -0.184,   2.588 0.0 -0.798,
                                    3.059 0.0  -0.996,   3.529 0.0 -0.674,
                                    4.000 0.0   0.000,
                              ]
                  scale       [     0.050 0.020,  0.200 0.100,
                                    0.400 0.150,  0.300 0.300,
                                    0.300 0.300,  0.300 0.300,
                                    0.300 0.300,  0.300 0.300,
                                    0.300 0.300,  0.300 0.300,
                                    0.290 0.290,  0.290 0.290,
                                    0.290 0.290,  0.280 0.280,
                                    0.280 0.280,  0.250 0.250,
                                    0.200 0.200,  0.100 0.100,
                                    0.050 0.050,
                        ]
                  }
            }
      }

# Sensor de Tiempo
DEF Tiempo TimeSensor {
      cycleInterval 4.0
      loop TRUE
      }

# Sensor de coordenadas
DEF Gusano CoordinateInterpolator {
      key [ 0.0, 0.25, 0.50, 0.75, 1.0 ]
```

Programa 7.12: **Animación** con sensores (Parte **1**/2).

```
      keyValue [                          # Posición en tiempo = 0.0
                -4.100 0.0  0.000,   -4.000 0.0  0.000,
                -3.529 0.0  0.674,   -3.059 0.0  0.996,
                -2.588 0.0  0.798,   -2.118 0.0  0.184,
                -1.647 0.0 -0.526,   -1.176 0.0 -0.962,
                -0.706 0.0 -0.895,   -0.235 0.0 -0.361,
                0.235 0.0   0.361,    0.706 0.0  0.895,
                1.176 0.0   0.962,    1.647 0.0  0.526,
                2.118 0.0  -0.184,    2.588 0.0 -0.798,
                3.059 0.0  -0.996,    3.529 0.0 -0.674,
                4.000 0.0   0.000,       # Posición en tiempo = 0.25
                -4.100 0.0 -1.000,   -4.000 0.0 -1.000,
                -3.529 0.0 -0.739,   -3.059 0.0 -0.092,
                -2.588 0.0  0.603,   -2.118 0.0  0.983,
                -1.647 0.0  0.850,   -1.176 0.0  0.274,
                -0.706 0.0 -0.446,   -0.235 0.0 -0.932,
                0.235 0.0  -0.932,    0.706 0.0 -0.446,
                1.176 0.0   0.274,    1.647 0.0  0.850,
                2.118 0.0   0.983,    2.588 0.0  0.603,
                3.059 0.0  -0.092,    3.529 0.0 -0.739,
                4.000 0.0  -1.000,       # Posición en tiempo = 0.5
                -4.100 0.0  0.000,   -4.000 0.0  0.000,
                -3.529 0.0 -0.674,   -3.059 0.0 -0.996,
                -2.588 0.0 -0.798,   -2.118 0.0 -0.184,
                -1.647 0.0  0.526,   -1.176 0.0  0.962,
                -0.706 0.0  0.895,   -0.235 0.0  0.361,
                0.235 0.0  -0.361,    0.706 0.0 -0.895,
                1.176 0.0  -0.962,    1.647 0.0 -0.526,
                2.118 0.0   0.184,    2.588 0.0  0.798,
                3.059 0.0   0.996,    3.529 0.0  0.674,
                4.000 0.0   0.000,       # Posición en tiempo = 0.75
                -4.100 0.0  1.000,   -4.000 0.0  1.000,
                -3.529 0.0  0.739,   -3.059 0.0  0.092,
                -2.588 0.0 -0.603,   -2.118 0.0 -0.983,
                -1.647 0.0 -0.850,   -1.176 0.0 -0.274,
                -0.706 0.0  0.446,   -0.235 0.0  0.932,
                 0.235 0.0  0.932,    0.706 0.0  0.446,
                 1.176 0.0 -0.274,    1.647 0.0 -0.850,
                 2.118 0.0 -0.983,    2.588 0.0 -0.603,
                 3.059 0.0  0.092,    3.529 0.0  0.739,
                 4.000 0.0  1.000,       # Posición en tiempo = 1.0
                -4.100 0.0  0.000,   -4.000 0.0  0.000,
                -3.529 0.0  0.674,   -3.059 0.0  0.996,
                -2.588 0.0  0.798,   -2.118 0.0  0.184,
                -1.647 0.0 -0.526,   -1.176 0.0 -0.962,
                -0.706 0.0 -0.895,   -0.235 0.0 -0.361,
                0.235 0.0   0.361,    0.706 0.0  0.895,
                1.176 0.0   0.962,    1.647 0.0  0.526,
                2.118 0.0  -0.184,    2.588 0.0 -0.798,
                3.059 0.0  -0.996,    3.529 0.0 -0.674,
                4.000 0.0   0.000,
                ]
      }
ROUTE Tiempo.fraction_changed TO Gusano.set_fraction
ROUTE Gusano.value_changed TO Cuerpo.set_spine
```

Programa 7.12: **Animación** con sensores (Parte **2**/2).

7.8 Texto fijo

Los **sensores de proximidad** pueden emplearse para poner información referente al mundo, que en todo momento este disponible para el usuario. El empleo más común para este tipo de aplicación de los sensores es la creación de menús, cuadros de información, objetos inmóviles, etc. En el **Programa 7.13** se observa la forma de hacerlo (**Figura 7.10**).

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

DEF Sensor_Global ProximitySensor {
      center      0 0 0
      size        10000 10000 10000
      }

DEF Invertir Transform {
      children    [
      Transform {
                  children [
                  Shape {
                              appearance Appearance {
                                    material Material { diffuseColor 1 1 0 }
                              }
                              geometry Text {
                                    string      "Puede ser cualquier nodo"
                                    fontStyle FontStyle {
                                          justify     "Left"
                                          family      "Arial"
                                          style       "BOLD"
                                          size        0.1
                                    }
                              }
                        }
                  ]
            translation  -0.7 -.38 -1
            }
      ]
}

ROUTE Sensor_Global.position_changed TO Invertir.set_translation
ROUTE Sensor_Global.orientation_changed TO Invertir.set_rotation
```

Programa 7.13: **Texto fijo** con sensores.

Figura 7.10: **Texto fijo** con sensores.

Capítulo 8: Sonido

El sonido en **VRML** puede ser **espacial** o no. Por **espacial** se entiende que hay una fuente emisora de sonido y que el observador percibe la posición de esta fuente en función de las posiciones relativas entre la fuente sonora y el observador **[Alarcón 00** y **VRML 97]**.

8.1 Sonido ambiental

El sonido que no tiene la calidad de espacial, es un sonido que se escucha siempre y siempre igual, se esté en la posición que se esté dentro del entorno. Se llama **sonido ambiente**, haciendo referencia al hecho que siempre está presente y crea una especie de ambientación global.

Para definir un sonido en **VRML**, se utiliza el nodo **Sound**. Para que sea un sonido ambiente, hace falta darle un rango de actuación muy grande (tan grande como lo sea nuestro entorno).

En el **Programa 8.1** tenemos un ejemplo de cómo definir un sonido ambiental (**Figura 8.1**).

```
#VRML V2.0 utf8

Shape {
      geometry Sphere { radius 2 }
      appearance Appearance { material Material { diffuseColor 1 1 0 } }
      }

Sound {
      minFront          200
      minBack           200
      maxFront          200
      maxBack           200
      spatialize        FALSE
      source AudioClip {
            url         "fondo.wav"
            startTime   1
            loop        TRUE
            }
      }
```

Programa 8.1: Sonido **ambiental**.

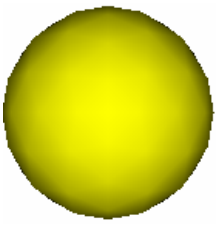


Figura 8.1: Esfera en el centro del mundo.

El ejemplo supone que en el directorio donde esté el mundo **VRML** hay un archivo **`fondo.wav`** con el sonido ambiente en formato **`WAV`**.

En el ejemplo aparecen varios conceptos nuevos. Primero, el hecho de que el sonido siempre tiene una **fuente** (**`source AudioClip`**). Para decir qué sonido queremos generar, hace falta definir un **`source AudioClip`** como a **fuente sonora**. En este, en **`AudioClip`** se ha de definir el **archivo de sonido** que queremos integrar al entorno (**`url`**) y, en este caso, definimos que vaya repitiéndose indefinidamente (**`loop`**).

Una vez definido el archivo de sonido, hace falta establecer las características de este sonido. Hemos dicho que el sonido ambiente no es espacial y, por lo tanto, ponemos a **FALSE** esta propiedad (**spatialize**). Los otros cuatro campos describen el **rango de acción del sonido**, forman una esfera de sonido; donde se escucha el sonido si se está dentro de la esfera y no se escucha si se está fuera. Esta esfera la hemos definido con un **radio** de **200** unidades.

8.2 Sonido espacial

El **sonido espacial** se basa en el hecho de que tenemos dos oídos y por lo tanto se manifiesta un desfase de tiempo de llegada a uno a otro oído del sonido producido por una fuente sonora. De esta forma, podemos detectar de donde proviene el sonido. Ahora bien, éste es el efecto percibido, pero lo que nos interesa es saber cómo se generan los sonidos y cómo funciona el modelo establecido en **VRML 2.0**.

Para comenzar una fuente sonora tiene una posición en el espacio desde donde salen las ondas sonoras. Un sonido normalmente no se emite en todas las direcciones con la misma intensidad. Por lo tanto, tiene una dirección privilegiada. Este centro y esta dirección, generan distintas intensidades donde el sonido se percibe diferente según la posición del usuario. Estas intensidades no son esféricas por el hecho de que hay una dirección privilegiada. Por tanto, son lo que conocemos como **elipsoides** (esferas alargadas). Finalmente el sonido pierde Intensidad conforme se aleja de su fuente de origen (fenómeno conocido como **atenuación**). Por tanto, el oyente va notando como baja el volumen del sonido conforme se aleja del lugar donde se ha producido. Pero en tanto que hay una dirección principal del sonido, la atenuación es más rápida ó notoria en la dirección opuesta a la dirección principal.

El valor **maxBack** y **maxFront** son las distancias máximas hacia **atrás** y hacia **adelante** (respectivamente) a la que se llega a escuchar el sonido. Más arriba ya no se escucha en absoluto.

En el **Programa 8.2** se observa como se monta una fuente sonora, sobre una esfera, la cual gira alrededor de otra esfera, para que sea posible apreciar el sonido espacial producido por la esfera satélite, al pasar frente al espectador de la escena (**Figura 8.2**).

Figura 8.2: Esfera como **foco del sonido**.

```
#VRML V2.0 utf8
# Definición de Sonido Espacial.

NavigationInfo { type "FLY" }

DEF Espacial Transform {
      children Transform {
            translation 10 0 0
            children [
                  Shape {
                        geometry Sphere { radius 0.2 }
                        appearance Appearance {
                              material Material { diffuseColor 1 1 0 }
                              }
                        }
                  Sound {
                        minFront          0.2
                        minBack           0.2
                        maxFront          20
                        maxBack           20
                        spatialize        TRUE
                        source AudioClip {
                              url         "zum.wav"
                              startTime   1
                              loop        TRUE
                              }
                        }
                  ]
            }
      }

DEF Cap Shape {
      geometry Sphere { radius 1 }
      appearance Appearance {
            material Material { diffuseColor 1 0.7 0.6 }
            }
      }

DEF MotorEspacial TimeSensor {
      startTime 1
      loop TRUE
      cycleInterval 2
      }

DEF MovimentEspacial OrientationInterpolator {
      key         [       0,          0.5,            1 ]
      keyValue    [ 0 1 0 0, 0 1 0 3.1416, 0 1 0 6.2832 ]
      }

ROUTE MotorEspacial.fraction_changed TO MovimentEspacial.set_fraction
ROUTE MovimentEspacial.value_changed TO Espacial.rotation
```

Programa 8.2: Sonido espacial.

Capítulo 9: Nivel de Detalle

El ajuste en el nivel de detalle en ambientes virtuales, nos permite que los mundos creados sean fáciles de navegar, ocultando los objetos que no se encuentran en nuestro alcance visual **[Alarcón 00** y **VRML 97]**.

9.1 Nivel de Detalle (LOD)

El **nivel de detalle** (**LOD: Level Of Detail**) es una técnica imprescindible en **gráficos 3D** generados en tiempo real y utilizado para poder mantener la velocidad de refresco de la pantalla sin gastar tiempo inútilmente. De lo que se trata es de tener modelos simplificados (en diferentes grados) de un mismo objeto. Entonces se elige cual de los modelos se enseñará en cada momento, según la distancia a la que está el observador. Esto se hace porque no tiene sentido dibujar un objeto muy complejo con centenares de polígonos, si este objeto ocupa **3x3 píxeles** en la pantalla. Si no tuviéramos modelos simplificados estaríamos malgastando tiempo en dibujar con todo detalle una cosa que finalmente no se apreciará.

La forma en que se trabaja al aplicar el **LOD** es a base de definir espacios o rangos delante del observador y asignar a uno de los modelos el nivel de detalle adecuado para ser visto a esas distancias. El nodo que **VRML** implementa para conseguir este efecto es precisamente el **LOD**.

El **Programa 9.1** muestra el efecto de irnos acercando a una silla. Conforme nos vamos acercando, el modelo va ganando en definición (**Figura 9.1**).

```
#VRML V2.0 utf8
LOD   {
      range [ 10 15 20 ]
      level [
            Group {                     # Nivel 1 (máximo nivel):
                  children [            # Dos cajas y cuatro cilindros
                        Transform {     # Respaldo
                              translation 0 3.5 -1.75
                              children Shape {
                                    geometry Box { size 3.5 3 0.5 }
                                    appearance Appearance {
                                          material Material {
                                                diffuseColor 0.8 0.6 0
                                                }
                                          }
                                    }
                              }
                        Shape {         # Base
                              geometry Box { size 3.5 0.5 3.5 }
                              appearance Appearance {
                                    material Material {
                                          diffuseColor 0.8 0.6 0
                                          }
                                    }
                              }
                        Transform {     # Pata posterior izquierda
                              translation -1.75 0 -1.75
                              children Shape {
                                    geometry Cylinder {
                                          radius 0.25
                                          height 10
                                          }
```

Programa 9.1: Definición de una silla con varios niveles de detalle (Parte **1**/4).

```
                        appearance Appearance {
                              material Material {
                                    diffuseColor 0.8 0 0.2
                                    }
                              }
                        }
                  }
            Transform {             # Pata posterior derecha
                  translation 1.75 0 -1.75
                  children Shape {
                        geometry Cylinder {
                              radius 0.25
                              height 10
                              }
                        appearance Appearance {
                              material Material {
                                    diffuseColor 0.8 0 0.2
                                    }
                              }
                        }
                  }
            Transform {             # Pata frontal izquierda
                  translation -1.75 -2.5 1.75
                  children Shape {
                        geometry Cylinder {
                              radius 0.25
                              height 5
                              }
                        appearance Appearance {
                              material Material {
                                    diffuseColor 0.8 0 0.2
                                    }
                              }
                        }
                  }
            Transform {             # Pata frontal derecha
                  translation 1.75 -2.5 1.75
                  children Shape {
                        geometry Cylinder {
                              radius 0.25
                              height 5
                              }
                        appearance Appearance {
                              material Material {
                                    diffuseColor 0.8 0 0.2
                                    }
                              }
                        }
                  }
            ]
      }
Group {                             # Nivel 2: seis cajas
      children [
            Transform {             # Respaldo
                  translation 0 3.5 -1.75
                  children Shape {
```

Programa 9.1: Definición de una silla con varios niveles de detalle (Parte **2**/4).

```
                  geometry Box { size 3.5 3 0.5 }
                  appearance Appearance {
                        material Material {
                              diffuseColor 0.8 0.6 0
                              }
                        }
                  }
            }
      Shape {                 # Base
            geometry Box { size 3.5 0.5 3.5 }
            appearance Appearance {
                  material Material {
                        diffuseColor 0.8 0.6 0
                        }
                  }
            }
      Transform {             # Pata posterior izquierda
            translation -1.75 0 -1.75
            children Shape {
                  geometry Box { size 0.5 10 0.5 }
                  appearance Appearance {
                        material Material {
                              diffuseColor 0.8 0 0.2
                              }
                        }
                  }
            }
      Transform {             # Pata posterior derecha
            translation 1.75 0 -1.75
            children Shape {
                  geometry Box { size 0.5 10 0.5 }
                  appearance Appearance {
                        material Material {
                              diffuseColor 0.8 0 0.2
                              }
                        }
                  }
            }
      Transform {             # Pata frontal izquierda
            translation -1.75 -2.5 1.75
            children Shape {
                  geometry Box { size 0.5 5 0.5 }
                  appearance Appearance {
                        material Material {
                              diffuseColor 0.8 0 0.2
                              }
                        }
                  }
            }
      Transform {             # Pata frontal derecha
            translation 1.75 -2.5 1.75
            children Shape {
                  geometry Box { size 0.5 5 0.5 }
                  appearance Appearance {
                        material Material {
                              diffuseColor 0.8 0 0.2
                              }
```

Programa 9.1: Definición de una silla con varios niveles de detalle (Parte **3**/4).

```
                                    }
                              }
                        }
                  ]
            }
      Group {                          # Nivel 3: dos cajas
            children [
                  Transform {          # Respaldo
                        translation 0 2.5 -1.75
                        children Shape {
                              geometry Box { size 4 5 0.5 }
                              appearance Appearance {
                                    material Material {
                                          diffuseColor 0.8 0.6 0
                                          }
                                    }
                              }
                        }
                  Transform {          # Base
                        translation 0 -2.5 0
                        children Shape {
                              geometry Box { size 4 5 4 }
                              appearance Appearance {
                                    material Material {
                                          diffuseColor 0.8 0.6 0
                                          }
                                    }
                              }
                        }
                  ]
            }
      Shape {                          # Nivel 4 (más bajo): una caja
            geometry Box { size 4 10 4 }
            appearance Appearance {
                  material Material { diffuseColor 0.8 0.6 0 }
                  }
            }
      ]
}
```

Programa 9.1: Definición de una silla con varios niveles de detalle (Parte **4**/4).

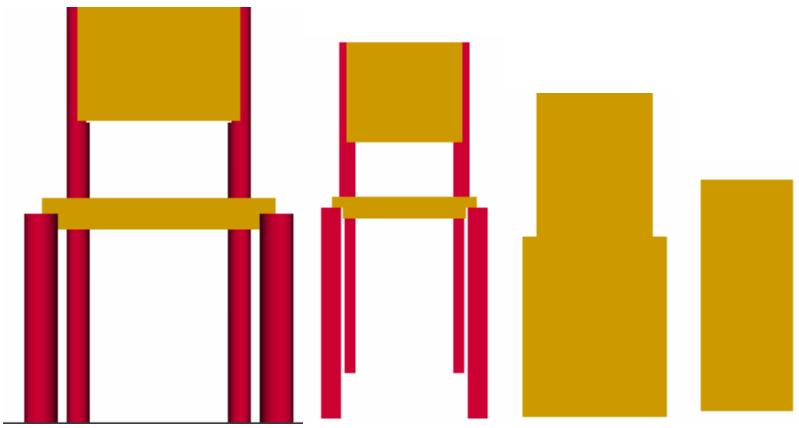


Figura 9.1: Definición de una silla con varios niveles de detalle.

Las partes del nodo **LOD** son las siguientes: en primer lugar, encontramos el campo **range** donde definimos las particiones de distancia donde se verá cada nivel de modelo. Queremos definir cuatro modelos diferentes y por eso definimos tres valores. El primer valor nos indica que se habrá de ver el modelo más detallado si el observador está a menos de **10 unidades** del objeto. El segundo valor nos indica que se ha de ver el segundo nivel de detalle si el observador está a menos de **15 unidades** (y a más de **10 unidades**) del objeto. El tercer valor nos indica que se ha de ver el tercer nivel de detalle si el observador está a menos de **20 unidades** (y a más de **15 unidades**) del objeto. Finalmente, no hace falta poner el cuarto valor porque ya se deduce que se ha de ver el cuarto nivel de detalle si el observador está a más de **20 unidades** del objeto.

El segundo elemento que encontramos es el campo **level**, que es donde ponemos las diferentes versiones del objeto en orden de más a menos detalle.

Evidentemente, los rangos que hemos escogido no producen un efecto suave de cambio, y son solamente de tipo didáctico.

9.2 Información de Navegación

El nodo **NavigationInfo** describe las características físicas del avatar (observador) y el tipo de controles que el navegador le ha de presentar. Contiene los siguientes campos:

- **avatarSize**.- especifica las dimensiones físicas del usuario en el mundo, con el propósito de detección de colisiones y seguimiento del terreno.
- **headlight**.- permite apagar o encender una luz direccional que surge de la cabeza del usuario.
- **speed**.- especifica la velocidad de movimiento de la escena en metros por segundo.
- **type**.- especifica el tipo de controles del navegador:
  - **ANY**.- aparece el tipo de controles por omisión del navegador, pero permite que el estilo de navegación cambie dinámicamente.
  - **WALK**.- presenta los controles para explorar el mundo a pie, como si se avanzara en un vehículo sobre el suelo.
  - **FLY**.- es similar a la anterior, solo que la gravedad puede ser ignorada.
  - **EXAMINE**.- se emplea para ver objetos de forma detallada.
  - **NONE**.- aparece el tipo de controles por omisión del navegador.

Capítulo 10: JavaScript

Aunque **VRML 2.0** da muchos mecanismos para definir interacción y comportamientos, hay cosas que no se pueden hacer directamente y entonces hace falta utilizar la potencia de un lenguaje de programación externo **[Alarcón 00** y **VRML 97]**. Esto se consigue a través del nodo **`Script`** y los lenguajes que se pueden utilizar son **Java** y **JavaScript** (entre otros). En este capítulo nos concentraremos en la descripción de la interacción entre **VRML** y **JavaScript**.

10.1 Nodo Script

El nodo **`Script`** permite las siguientes acciones:

- **Captar los eventos** que necesitamos.
- Conseguir los **valores** que forman los eventos y procesarlos.
- Enviar los **resultados** de este procesamiento como **evento de salida** (para ser encaminados, mediante **`ROUTE`**, hacia otros nodos).

Un nodo **`Script`** está formado por dos partes principales: las **definiciones de campos** y **eventos**, y el **código en el lenguaje** que se haya elegido.

Es importante observar que en esté nodo podemos definir campos y eventos según nuestras necesidades, en contraste con los otros nodos de **VRML** que ya tienen predefinidos todos sus componentes.

El lugar donde se pone el código del lenguaje es en el campo **`url`**. Este campo permite escribir todo el código entre comillas dobles (**“”**) o bien referenciar a un archivo externo donde figure todo el código.

A continuación vemos un esquema de la estructura del nodo **`Script`**:

```
Script {
      Campo…
      …
      Campo…
      …
      eventIn…
      …
      eventIn…
      …
      eventOut…
      …
      eventOut …
      url "javascript:
            función X (v,t) {
                  …
                  }
            …
            función Z (v,t) {
                  …
                  }
            “
      }
```

Aquí se pueden observar las dos partes que definimos arriba. Primero se encuentran las **definiciones de campos** y **eventos**. Estos no han de estar ordenados de ninguna manera concreta. En segundo lugar, encontramos el campo **`url`** donde, entre comillas, se define el tipo de código que se utiliza (**`JavaScript`**) y, a continuación, todas las funciones que configuran nuestros procesos.

10.2 Eventos y funciones JavaScript

Hay una relación directa entre los **nombres de los eventos** y los **nombres de las funciones** del código. Esta relación se establece para que, cuando llegue un evento de entrada al nodo **`Script`** en cuestión, él llame a la función que tiene el mismo nombre que el evento de entrada referenciado y así se pueda capturar el valor que ha de recibir y procesarlo.

El mecanismo implementado por el nodo **`Script`** hace que toda función de **JavaScript** asociada a un evento de entrada tenga dos parámetros por defecto: el **valor recibido** por el **`eventIn`** y el **instante de tiempo** en que se ha generado dicho evento, de está forma, la función puede entonces utilizar el valor del evento que ha recibido e incluso discernirlo de otros gracias al hecho de que también dispone del tiempo en que dicho evento fue generado.

A continuación, damos un ejemplo donde un **sensor de proximidad** envía un evento de **cambio de posición** del punto de vista de un **`Script`** que determina si la coordenada **`x`** de este punto es más grande que **`5`** (**Programa 10.1**).

```
#VRML V2.0 utf8

DEF SensorPuntoVista ProximitySensor {
      size 100 100 100
      }

DEF SiguePuntoVista Script {
      eventIn SFVec3f nuevaPosicion
      eventOut SFTime activaAlarma
      eventOut SFTime apagaAlarma
      url "javascript:
            function nuevaPosicion (v,t) {
                  if (v[0] > 5)
                        activaAlarma = t;
                  else
                        apagaAlarma = t;
                  }
            "
      }

DEF ClipAlarma AudioClip {
      url         "alarma.wav"
      startTime   -1
      loop        TRUE
      }

Sound {
      source      USE ClipAlarma
      maxFront    200
      maxBack     200
      }

ROUTE SensorPuntoVista.position_changed TO SiguePuntoVista.nuevaPosicion
ROUTE SiguePuntoVista.activaAlarma TO ClipAlarma.startTime
ROUTE SiguePuntoVista.apagaAlarma TO ClipAlarma.stopTime
```

Programa 10.1: Nodo **`Script`** asociado a un **sensor de proximidad**.

Analicemos este código parte por parte:

- En primer lugar, tenemos la definición del **ProximitySensor** al cual le damos un nombre para referenciarlo en el **ROUTE**.
- Después encontramos el **Script** al cual también le damos un nombre. Este nodo no tiene ningún campo o evento definido a priori aparte del campo **url**. Por tanto, hace falta que nosotros definamos los campos y eventos que nos interesen. De momento, como la única cosa que queremos hacer es captar el evento de cambio de posición del **ProximitySensor**, definimos un **eventIn** del tipo **Vector3D** univaluado en coma flotante (**SFVec3f**, **S**ingle-valued **F**ield of **Vec**tor **3** Component in **f**loating point). A este evento de entrada le damos el nombre **nuevaPosicion**, lo cual implica que tendremos que definir una función de **JavaScript** con el mismo nombre para poder captar el valor que entre para este evento.
- En el campo **url** encontramos el código **JavaScript**. De momento, sólo se define una sola función, la que ha de ir asociada al **evento de entrada**. Así pues, definimos una función que tiene por nombre **nuevaPosicion** y que, por defecto, **VRML** le define en dos parámetros: **v**, que es el **valor** que lleva al **eventIn** y **t**, que es el instante de **tiempo** en que se ha generado el valor. De esta forma, desde dentro de la función podemos acceder al valor del evento de entrada y podemos comparar la coordenada **x** con nuestro límite de **5** unidades.
- Al final se encuentra **ROUTE** con el cual enlazamos el **eventOut position_changed** que tiene predefinido el **ProximitySensor** con el **eventIn nuevaPosicion** que hemos definido en nuestro nodo **Script**.

Así como en **VRML** no se accede nunca a los componentes de los datos que pertenecen a tipos no escalados con **SFVec3f**, **SFRotation**, **SFColor**, **SFVec2f** y todos los **MF**, cuando se programan puede interesarnos acceder a estos componentes. En este caso, un valor de uno de estos tipos se comporta como si fuese un arreglo de **javaScript** y por lo tanto, se accede a los componentes indexando desde **0** (cero) hasta el número de componentes menos **1**. Por ejemplo: a un campo **SFColor** con nombre **ColorMio**, se le podría asignar los valores **RGB (1 0.5 0.3)** desde un **Script** indexando de la siguiente forma: **ColorMio[0]=1**, **ColorMio[1]=0.5** y **ColorMio[2]=0.3**.

10.3 Eventos

Los **eventOut** se definen en la primera parte del nodo **Script** de forma similar a los **eventIn**. Para poder generar el evento de salida a través del **eventOut** que hemos definido, sólo hace falta asignarle un valor desde dentro de la función que ha de generar el evento.

En el ejemplo anterior suena una alarma cuando detectamos que el punto de vista ha sobrepasado el límite de **5** unidades en el **Eje X** (**Programa 10.1**). Los elementos necesarios son los siguientes:

- Se añaden dos **eventOut** a nuestro **Script**: **activaAlarma** y **apagaAlarma**. De esta forma se activa o se apaga, según la posición del usuario. Estos eventos de salida son del tipo **SFTime** porque se tiene un tiempo de inicio y el tiempo de finalización del sonido.
- En el código de **javaScript**, se añade una acción a realizar al **if** y hemos añadido una opción al **else**. Lo que hacemos es aprovechar el hecho que toda función asociada a un **eventIn** tiene dos parámetros por defecto: el **valor** del evento y el **tiempo** en que se ha generado. Así pues en **t** tenemos el instante de tiempo en que se ha generado el cambio de posición del usuario y, por lo tanto, es el instante indicado para activar o apagar el sonido. Una vez que el **if** ha diferenciado si el usuario ha traspasado el límite, se asigna el tiempo, al evento de salida correspondiente.

Es importante entender que el evento de salida se genera por el solo hecho de asignar un valor al **eventOut**. El momento en que se envían estos eventos de salida, es cuando **VRML** ha acabado de ejecutar todo el **Script**.

10.4 Campos

Los **campos** también se definen en la primera parte del nodo **Script** de forma similar a los **eventIn** y a los **eventOut**. Los **campos** sirven como **variables globales** para el **Script** y para guardar los valores a lo largo de toda la ejecución del entorno. Con esto podemos comparar valores de eventos nuevos con valores antiguos que hayamos guardado en los campos.

Lo que haremos ahora es que suene la alarma sólo la primera vez que el usuario pasa el límite. Si entonces el usuario vuelve hacia atrás y después vuelve a pasar el límite, la alarma ya no ha de sonar (**Programa 10.2**).

```
#VRML V2.0 utf8

DEF SensorPuntoVista ProximitySensor {
      size 100 100 100
      }

DEF SiguePuntoVista Script {
      eventIn  SFVec3f nuevaPosicion
      field    SFBool  haEntrado FALSE
      eventOut SFTime  activaAlarma
      eventOut SFTime  apagaAlarma

      url "javascript:
            function nuevaPosicion (v,t) {
                  if (v[0] > 5)
                        if(!haEntrado){
                              activaAlarma = t;
                              haEntrado = TRUE;
                        }
                  else  apagaAlarma = t;
                  }
            "
      }

DEF ClipAlarma AudioClip {
      url         "alarma.wav"
      startTime   -1
      loop        TRUE
      }

Sound {
      source      USE ClipAlarma
      maxFront    200
      maxBack     200
      }

ROUTE SensorPuntoVista.position_changed TO SiguePuntoVista.nuevaPosicion
ROUTE SiguePuntoVista.activaAlarma TO ClipAlarma.startTime
ROUTE SiguePuntoVista.apagaAlarma TO ClipAlarma.stopTime
```

Programa 10.2: Nodo **Script** en donde suena una alarma una sola vez.

Se añadieron los elementos siguientes:

- Un campo de tipo **SFBool** con el nuevo nombre **haEntrado**. A este campo le hemos dado un valor inicial **FALSE** porque al iniciar, el usuario aún no ha entrado nunca a la zona prohibida. Todo campo ha de tener un valor inicial independientemente de la utilización que se haga de él.

- Un **if** dentro de la rama verdadera del primero, poniéndole una condición para activar la alarma, se ha de cumplir que el usuario nunca haya pasado el límite. Esto se consigue comprobando si el campo **haEntrado** es falso o verdadero.
- A la rama verdadera se le coloca la acción de asignar el valor **TRUE** al campo **haEntrado**, para tener conocimiento de que el usuario ya ha traspasado una vez el límite.
- No hace falta hacer nada más porque cada vez que se llame el **Script** el campo que hemos definido tendrá el último valor que le hayamos asignado.

10.5 Campo mustEvaluate

Inicialmente hemos dicho que, a parte del campo **url** y el nodo **Script** no tenía ningún otro campo predefinido. Sin embargo, el nodo **Script** dispone de dos campos predefinidos: **mustEvaluate** y **directOutput**.

Durante la ejecución de un entorno **VRML**, el navegador tiene la autorización (por especificación) de gestionar los eventos en el momento que le sea más idóneo. Esto puede provocar que se acumulen una serie de eventos durante un lapso de tiempo y que de golpe sean todos evaluados (evidentemente, en el orden que se han generado).

Así, cuando programamos el **Script** para gestionar varios eventos, puede pasar que no se nos evalúe el **Script** cada vez que se genera el evento de entrada que necesitamos. En este caso pasaría que, al cabo de un rato, se evaluaría nuestro **Script** tantas veces como eventos de entrada se hayan acumulado.

Para controlar esto, el nodo **Script** dispone del campo predefinido **SFBool mustEvaluate**. Por defecto, este campo tiene el valor **FALSE**, cosa que significa que la evaluación del **Script** puede ser pospuesta. Si queremos que nuestro **Script** se evalúe cada vez que se genere un evento de entrada de los que gestionamos, entonces hace falta poner el campo **mustEvaluate TRUE**.

Hace falta tener en cuenta que poner **mustEvalunte TRUE** implica un control más exhaustivo al navegador y, por lo tanto, se pierde eficiencia. Sólo debe hacerse cuando sea estrictamente necesario.

10.6 Acceso a otros nodos y el campo directOutput

A veces es práctico poder acceder directamente a los **exposedField**, **eventOut** y **eventIn** de nodos externos al **Script**, sin tener que definir todo un conjunto de **ROUTE**. Esto da más control sobre el entorno, ya que así no dependemos de la gestión de eventos que hace el navegador. Podríamos definir un nodo de transformación con un cubo como geometría, del cual queremos saber el valor del campo de translación desde dentro del **Script** sin necesidad de que se genere un evento (**Programa 9.3**).

Analicemos paso por paso lo que se ha hecho en el **Programa 10.3**:

- Definimos un nodo de transformación al cual ponemos un nombre **TransfCub** para poder acceder a él. Este nodo contiene un cubo como geometría, y un **TouchSensor** para que el cubo sea sensible al tacto.
- A continuación definimos un **Script** con los siguientes elementos:
  - Un **eventIn SFTime click** que nos captura los eventos de que se ha activado el **TouchSensor**.
  - Definimos después un campo de tipo **nodo** (**SFNode**) para poder referenciar directamente el nodo de transformación externo. A este campo le llamamos **Tcub** y para dejar claro que no es un nodo nuevo, sino que en realidad estamos haciendo referencia al nodo de transformación, añadimos **USE TransfCub**. Esto quiere decir que se está utilizando el nodo **TransfCub** externo.
  - Entonces definimos la función asociada al **eventIn click**. En esta función comenzamos observando el valor de la componente **X** del campo de translación del nodo de transformación **TransfCub**. Esto lo conseguimos a través de la referencia que hemos montado en la primera parte del **Script**, es decir, a través del **TCub**.

```
#VRML V2.0 utf8

DEF TransfCub Transform {
      translation 0 0 0
      children [
            Shape {
                  geometry Box { size 1 1 1 }
                  appearance Appearance{
                        material Material{
                              diffuseColor 0 1 1
                              }
                        }
                  }
            DEF TS TouchSensor {}
            ]
      }

DEF MeuScript Script {
      directOutput      TRUE
      eventIn SFTime    click
      field   SFNode    TCub USE TransfCub

      url "javascript:
            function click(v,t) {
                  if(TCub.translation[0] > 5) TCub.translation[0] = 0;
                  else TCub.translation[0] = TCub.translation[0] + 1;
                  }
            "
      }

ROUTE TS.touchTime TO MeuScript.click
```

Programa 10.3: Acceso directo a un nodo externo a un **Script**, con modificación de posición.

En el **Programa 10.3** vemos como podemos acceder a información de nodos externos sin tener que establecer **ROUTE** que nos encaminen eventos. Pero de está forma, sólo podemos acceder a los **exposedField** y a los **eventOut** para leerlos.

Para acceder a los **exposedField** o a los **eveutIn** para modificarlos, no podríamos, a menos que utilicemos el campo que suministran los **Script**, **field**, **SFBool directOutput**. Este campo por defecto tiene el valor **FALSE**. Así, para poder tener acceso directo a los campos de nodos externos y poderlos modificar, hace falta ponerlo a **TRUE**.

El **Programa 10.4** y la **Figura 10.1** muestran una esfera describiendo una función matemática empleando **javaScript**.

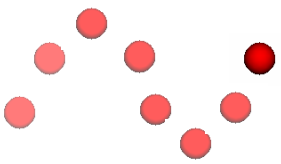


Figura 10.1: Esfera describiendo una función seno.

```
#VRML V2.0 utf8
Group {
      children [
            DEF MoveMe Transform {
                  children Shape {
                        appearance Appearance {
                              material Material {
                                    diffuseColor 1 0 0
                                    }
                              }
                        geometry Sphere { radius 0.1 }
                        }
                  },
            DEF Clock TimeSensor {
                  cycleInterval     4.0
                  loop              TRUE
                  startTime         2.0
                  stopTime          0.0
                  },
            DEF Mover Script {
                  field       SFFloat cycles 2.0
                  eventIn     SFFloat set_fraction
                  eventOut    SFVec3f position_changed
                  url "javascript:
                        function set_fraction( f, tm ) {
                              position_changed[0] = f;
                              position_changed[1] =
                  Math.sin( ((f*360)*cycles)/180.0 * 3.1415927 );
                              position_changed[2] = 0.0;
                        }"
                  }
            ]
      }

ROUTE Clock.fraction_changed TO Mover.set_fraction
ROUTE Mover.position_changed TO MoveMe.set_translation
```

Programa 10.4: Esfera describiendo una función seno.

10.7 Fractales

Un aspecto interesante en la modelación, es el empleo de figuras fractales, las cuales permiten dibujar objetos complejos, repitiendo modelos simples. Un ejemplo de lo que se puede obtener se muestra en el **Programa 10.5** y en la **Figura 10.2**, en los cuales se muestra la creación de un cactus mediante el uso de reglas escritas en **JavaScript**.

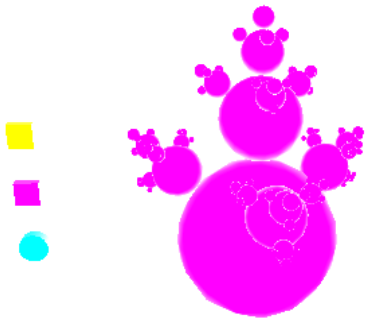


Figura 10.2: Cactus dibujado por procedimientos fractales.

```
#VRML V2.0 utf8

# Mundo inicial vacio
DEF ROOT Transform{
      translation 0 -1 0 scale 1.5 1.5 1.5
      children[]
      }

# JavaScript que genera el fractal
DEF Cactus Script {
  eventIn   SFBool Mas
  eventIn   SFBool Menos
  eventOut  MFNode Nuevo
  field     SFNode ROOT USE ROOT
  directOutput TRUE
url ["javascript:
function initialize(){
  MaxIt = 10;
  strArr = new MFString();  strArr.length = MaxIt;
  strArr[0] =
'DEF s0 Shape{ appearance Appearance{ material Material{ diffuseColor 0 1 0}} '+
'geometry Sphere{}}';
  for ( i= 1; i<MaxIt; i++)  strArr[i] =
'DEF s'+i+' Transform{ children[ '+
strArr[i-1] +
']} Transform{ translation 0 1.5 0 scale .5 .5 .5 children USE s'+ i +'} USE s0 '+
'DEF g'+i+' Transform{ rotation 1 0 0 1 translation 0 .72 1.12 '+
'scale .33 .33 .33 children USE s'+ i +'} '+
'Transform{ rotation 0 1 0 2.09 children USE g'+ i +'} '+
'Transform{ rotation 0 1 0 -2.09 children USE g'+ i +'}';
  iter = 3;
  Nuevo = Browser.createVrmlFromString( strArr[iter] );
  ROOT.addChildren = Nuevo;
}
// Incrementa el orden del fractal
function Mas(value){
 if (value && (iter < MaxIt-1)){
  iter++;
  ROOT.removeChildren = Nuevo;
  Nuevo = Browser.createVrmlFromString( strArr[iter] );
  ROOT.addChildren = Nuevo;
 }
}
// Decrementa el orden del fractal
function Menos(value){
 if (value && (iter>0)){
  ROOT.removeChildren = Nuevo;
  iter--;
  Nuevo = Browser.createVrmlFromString( strArr[iter] );
  ROOT.addChildren = Nuevo;
 }
}
"]}
```

Programa 10.5: Cactus fractal (Parte **1**/2).

```
# Controles
DEF Tiempo TimeSensor{ cycleInterval 20}
DEF Animacion OrientationInterpolator{ key [ 0, .5, 1 ]
 keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28 ]}

Transform{ translation -4 -1 0 rotation 1 0 0 1.57 children[
 Shape{ appearance Appearance{ material Material{ diffuseColor 1 0 0}}
  geometry Cylinder{ height .2 radius .25}}
 DEF tsRot TouchSensor{} ]}
Transform{ translation -4 1 0 children[
 Shape{ appearance Appearance{ material Material{ diffuseColor 0 0 1}}
  geometry Box{ size .4 .4 .2 }}
 DEF tsMas TouchSensor{} ]}
Transform{ translation -4 0 0 children[
 Shape{ appearance Appearance{ material Material{ diffuseColor 0 1 0}}
  geometry Box{ size .4 .4 .2 }}
 DEF tsMenos TouchSensor{} ]}

#Rutas
ROUTE tsMas.isActive TO Cactus.Mas
ROUTE tsMenos.isActive TO Cactus.Menos
ROUTE tsRot.touchTime TO Tiempo.startTime
ROUTE Tiempo.fraction_changed TO Animacion.set_fraction
ROUTE Animacion.value_changed TO ROOT.rotation
```

Programa 10.5: Cactus fractal (Parte **2**/2).

Capítulo 11: VRML y Java

11.1 VRML y Java

Indudablemente **VRML** es una poderosa herramienta de visualización, sin embargo, carece del poder y versatilidad de cálculo que proporcionan los lenguajes de propósito general **[Alarcón 00**, **Deitel 98** y **VRML 97]**. Para realizar la actualización de mundos con **VRML**, es necesario crear un nuevo programa **VRML** para cada caso específico, lo cual resulta engorroso. Para resolver este problema es posible hacer que **VRML** trabaje conjuntamente con un lenguaje de propósito general como **Java**, para que éste cree los objetos que pueden cambiar de posición, orientación, tamaño, etc. dentro de un mundo virtual.

El lenguaje **Java** es un lenguaje de alto nivel de propósito general (parecido a **C++**), orientado a objetos, dinámico (que cambia en tiempo de ejecución), con tipos de datos duros (el lenguaje prohíbe operaciones con datos que no sean del tipo al que están destinados), tiene comprobación estática de tipos (no sucede al momento de ejecución) y es concurrente (permite varios hilos de control).

Java fue desarrollado por **Sun Microsystems** en **1995** para la creación de páginas Web con un contenido dinámico. Los sistemas **Java** consisten en el entorno, el lenguaje, la interfaz de programación de aplicaciones (**API**, **A**pplications **P**rogramming **I**nterface) de **Java** y las bibliotecas de clases. Las clases son piezas predefinidas que se agrupan en categorías llamadas paquetes. **Java** esta disponible de forma gratuita a través de Internet en el sitio: **`www.javasoft.com`**.

En **Java** un documento dinámico se llama **applet**. Un programador crea un **applet** escribiendo un programa fuente en el lenguaje de programación **Java**, luego el compilador de **Java** traduce esté código en una representación de **código de bytes** (*bytecode*), el cual se carga en un servidor en la Web. Cuando el visualizador solicita una **URL** (**U**niform **R**esource **L**ocators: para localizar datos en **Internet**) relacionada con la **applet**, el servidor la ejecuta. El intérprete suministra el ambiente de ejecución de **Java** y el acceso a la pantalla del usuario y a **Internet**.

Java y **VRML** pueden interactuar, uniendo una poderosa herramienta de visualización (**VRML**) con las ventajas de un lenguaje de programación como **Java**. Para visualizar un mundo virtual en **VRML**, es necesario tener instalado (en el sistema), un navegador de **Internet** capaz de aceptar un visualizador de **VRML**. Para ejecutar programas en **Java** también es necesario instalar el intérprete correspondiente, un visualizador que ejecuta **Java** necesita tener un intérprete **HTML** y un intérprete para aplicaciones elementales.

Para lograr la integración de **Java** con **VRML**, es necesario utilizar la **External Authoring Interface** (**EAI**), la cual permite la comunicación y el paso de parámetros entre mundos virtuales **VRML** y los programas escritos en **Java**. Para poder compilar un programa en **Java** que utilice clases **VRML**, es necesario copiar el paquete de clases **VRML** de la **EAI**, como librería externa de **Java**. Por ejemplo para el visor de **VRML Cosmo Player 2.1**, el archivo que contiene los paquetes de clases se llama **`npcosmop21.jar`** y esta ubicado en (Windows):

```
C:\Archivos de programa\CosmoSoftware\CosmoPlayer\npcosmop21.jar
```

El archivo **`npcosmop21.jar`** tiene que ser copiado al directorio de **Java** **`C:\jdk1.3\jre\lib\ext\`**, sí **Java** se encuentra instalado en **`C:\jdk1.3`**.

Para realizar un programa en **Java** el cual permita la interacción con un ambiente virtual en **VRML**, son necesarios al menos los siguientes archivos de programa: **1)** el programa compilado en **Java**, **2)** un intérprete de **HTML** y **3)** un ambiente virtual de **VRML**.

Por ejemplo, veamos el código en **Java** de un programa capaz de agregar y remover objetos en un mundo virtual **VRML** (una esfera en este caso). En la primera parte se importan las clases de **Java** y **VRML**, luego se definen variables y se establece la ventana de controles y dialogo con el usuario, después se definen las variables para interactuar con el navegador y el visualizador de **VRML** (donde se tiene también el código **VRML** que será insertado en el mundo virtual) y al final se tiene que mediante eventos del ratón, se introducen las instrucciones de agregado y borrado de objetos en el mundo virtual (**Programa 11.1**).

```
/* Librerías que se incluyen                                          */
import java.awt.*;      /* Clases abstractas para manipulación gráfica */
import java.applet.*;   /* Applet para interactuar con el navegador    */
/* Clases de VRML, para interacción Java-VRML                         */
import vrml.external.field.*;
import vrml.external.Node;
import vrml.external.Browser;
import vrml.external.exception.*;

/* Applet que interactúa con el navegador                             */
public class vrmljava1 extends Applet {
      TextArea Salida = null;       /* Área de texto                  */
      boolean error = false;        /* Tipo de error                  */
      Browser Navegador;            /* Navegador a emplear            */
      Node root;                    /* Mundo virtual Vacío            */
      Node[] Dibujo;                /* Nodo para VRML                 */
      /* Eventos para agregar y remover un nodo en VRML               */
      EventInMFNode Agrega;         /* Evento de entrada a VRML       */
      EventInMFNode Quita;          /* Evento de entrada a VRML       */
      /* Inicia la ventana de controles y dialogo                     */
      public void init() {
            /* Agrega Botones                                         */
            add(new Button("Agregar Esfera"));
            add(new Button("Remover Esfera"));
            Salida = new TextArea(5, 40);
            add(Salida);
            }
      /* Inicia la rutina en el navegador                             */
      public void start() {
            /* Navegador en donde se despliega la información         */
            Navegador = (Browser) vrml.external.Browser.getBrowser(this);
            try {
                  /* Mundo donde se realizarán las operaciones        */
                  root = Navegador.getNode("ROOT");
                  Agrega =
                        (EventInMFNode) root.getEventIn("addChildren");
                  Quita =
                        (EventInMFNode) root.getEventIn("removeChildren");
                  /* Código VRML a insertar en el mundo virtual       */
                  Dibujo = Navegador.createVrmlFromString(
                              "Shape {\n" +
                              "  appearance Appearance {\n" +
                              "    material Material {\n" +
                              "      diffuseColor 1 1 0\n" +
                              "    }\n" +
                              "  }\n" +
                              "  geometry Sphere {}\n" +
                              "}\n");
                  }
            catch (InvalidNodeException e) {
                  error = true;
                  }
```

Programa 11.1: Programa **vrmljava1.java** (Parte **1**/2).

```
            catch (InvalidEventInException e) {
                  error = true;
                  }
            catch (InvalidVrmlException e) {
                  error = true;
                  }
            }
      /* Eventos de los botones                                          */
      public boolean action(Event event, Object what) {
            if (event.target instanceof Button) {
                  Button b = (Button) event.target;
                  if (b.getLabel() == "Agregar Esfera") {
                        Salida.appendText("Agregando Esfera...\n");
                        Agrega.setValue(Dibujo);
                        }
                  else if (b.getLabel() == "Remover Esfera") {
                        Salida.appendText("Removiendo esfera...\n");
                        Quita.setValue(Dibujo);
                        }
                  }
            return true;
            }
      }
```

Programa 11.1: Programa `vrmljava1.java` (Parte **2**/2).

Existen diversas operaciones que es posible realizar al insertar o remover objetos en un mundo virtual, como son agregar objetos, quitar objetos, cambiar la posición de los objetos, simular movimiento y hasta llegar a la animación de los objetos.

Para agregar un objeto desde **Java** para un mundo virtual en **VRML** se emplea el nodo `Agrega.setValue()` cuyo empleo se observa en el **Programa 11.1**, en donde se agrega el objeto ó los objetos contenidos en `Shape` al mundo virtual. De igual forma es posible quitar objetos con `Quita.setValue()`, en donde se remueven los objetos que se encuentran en `Shape`.

Las operaciones de cambiar la posición de los objetos, simular movimiento y llegar a la animación de los objetos se realizan mediante una secuencia de operaciones `Agrega.setValue()` y `Quita.setValue()` usadas de forma alternada.

Java incluye un compilador `javac`, el cual produce un código de bytes. La entrada es un archivo fuente en **Java** y produce archivos `*.class` de las clases publicas del programa en **Java**. Para el **Programa 11.1** se tiene la instrucción `javac vrmljava1` y produce la salida `vrmljava1.class`.

Una llamada a un **applet** se realiza a través de un navegador de **Internet** por medio de un programa en **HTML**, para mostrar el mundo virtual en **VRML** y ejecutar el **applet** de **Java**. La primera parte del **Programa 11.2**, muestra el espacio designado para mostrar el espacio virtual de **VRML** y la segunda establece el espacio de trabajo del **applet** de **Java**.

```
<html>

<head>
<title>Ejemplo 10.1: Uso de VRML con Java</title>
</head>

<body bicolor = "#000000" text = "#FFFFFF">

<p align = "center">
<embed src = "root.wrl" border = "0" height = "300" width = "500">
<applet code = "vrmljava1.class" width = "500" height = "120" mayscript>
</applet>
</p>

</body>
</html>
```

Programa 11.2: Código **HTML** para mostrar un mundo **VRML** y ejecutar un **applet** de **Java**.

El programa **HTML**, realiza una llamada a **VRML** y a un código de **Java**. El mundo virtual en **VRML**, puede contener cualquier cantidad de objetos o puede encontrarse vacío, sólo es necesario que permita insertar el código generado desde **Java**.

El **Programa 11.3**, muestra un ejemplo en donde es posible insertar un mundo virtual de cualquier tipo en la línea: `DEF ROOT Group {}`, por simplicidad se define un mundo vacío.

```
#VRML V2.0 utf8
NavigationInfo { type "EXAMINE" }

DEF ROOT Group {}
```

Programa 10.3: Programa `root.wrl` del mundo virtual vacío.

En la **Figura 11.1** se observa el resultado al ejecutar la página **HTML**, y cargar el mundo **VRML** y el **applet** de **Java**.

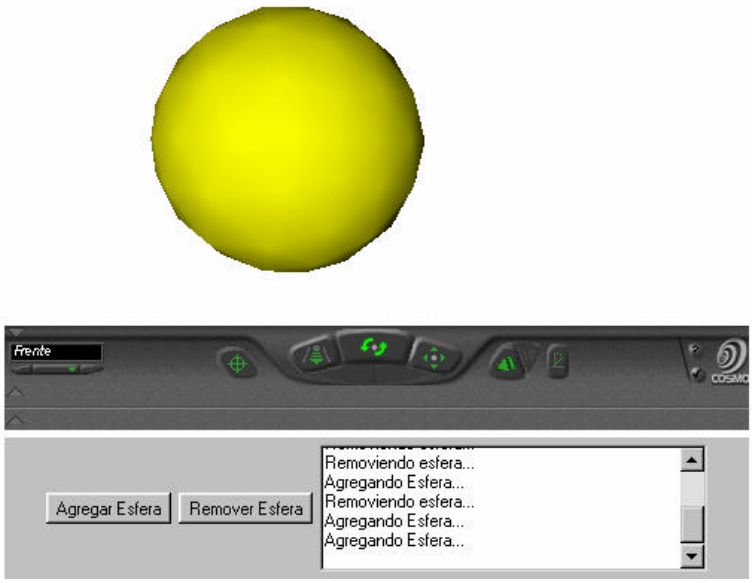


Figura 11.1: Interacción de **VRML** y **Java**.

En los **Programas 11.4**, **11.5**, **11.6** y **11.7** en **Java** se observan las rutinas con las que es posible agregar diferentes objetos y modificar sus características, comunicándole todas ellas mediante los eventos de **VRML** y **Java**.

```
/* Librerías que se incluyen                                              */
import java.awt.*;                /* Clases abstractas                    */
import java.applet.*;             /* Clase Applet                         */
import java.util.*;               /* Utilidades de Java                   */
/* Clases que permiten la interacción Java_VRML                           */
import vrml.external.field.*;
import vrml.external.exception.*;
import vrml.external.Node;
import vrml.external.Browser;
/* Clases para calcular puntos y manipularlos                             */
import V2Grupo;
import V2Rotacion;
import V2Vec3D;

/* Clase que interactúa con el navegador                                  */
public class vrmljava2 extends Applet {
/* Controles de transformación                                            */
Scrollbar transx; /* Traslado en X                                        */
Scrollbar transy; /* Traslado en Y                                        */
Scrollbar transz; /* Traslado en Z                                        */
Scrollbar rotx;   /* Rotación en X                                        */
Scrollbar roty;   /* Rotación en Y                                        */
Scrollbar rotz;   /* Rotación en Z                                        */
Scrollbar escx;   /* Escala en X                                          */
Scrollbar escy;   /* Escala en Y                                          */
Scrollbar escz;   /* Escala en Z                                          */
Scrollbar colr;   /* Color Rojo                                           */
Scrollbar colg;   /* Color Verde                                          */
Scrollbar colb;   /* Color Azul                                           */

Browser browser;  /* Navegador a usar                                     */
Node root;        /* Mundo virtual vacío                                  */
Vector grupo;     /* Objetos que se agregaran a la escena                 */
int indice;       /* Contador                                             */

V2Grupo Grupo_actual = null;  /* Grupo de objetos actual                  */
static int Rango_trans = 40;  /* Intervalo de transformaciones            */

/* Agrega botones                                                         */
public void Boton1(String nombre,GridBagLayout malla,
                   GridBagConstraints c) {
      Button boton = new Button(nombre);
      malla.setConstraints(boton, c);
      add(boton);
      }

/* Agrega etiquetas                                                       */
public void Etiqueta1(String nombre, GridBagLayout malla, GridBagConstraints c) {
      Label etiqueta = new Label(nombre, Label.CENTER);
      malla.setConstraints(etiqueta, c);
      add(etiqueta);
      }
```

Programa 11.4: Inserción y transformación de objetos con **Java** **`vrmljava2.java`** (Parte **1**/6).

```
/* Agrega barras de desplazamiento                                          */
public Scrollbar Barra1(GridBagLayout malla, GridBagConstraints c,
                        int orientacion,
                        int valor,
                        int min,
                        int max) {
      Scrollbar barra = new Scrollbar(orientacion, valor, 1, min, max);
      malla.setConstraints(barra, c);
      add(barra);
      return barra;
      }

/* Dibuja el Applet y carga la malla                                        */
public void init() {
      System.out.println("Inicio 1...");
      /* Variables para crear la Malla de controles                         */
      float Columna_1 = 0.04f;
      float Columna_3 = 0.04f;
      float Columna_2 = 1.0f - Columna_1 - Columna_3;
      /* Crea la malla                                                      */
      GridBagLayout malla = new GridBagLayout();
      GridBagConstraints c = new GridBagConstraints();
      setFont(new Font("Arial", Font.PLAIN, 12));
      setLayout(malla);
      /* Carga la malla con botones, etiquetas y barras                     */
      c.fill = GridBagConstraints.BOTH;
      c.gridwidth = 2;
      c.gridheight = 2;
      c.weightx = Columna_1;
      Boton1("Cubo", malla, c);
      c.gridheight = 1;
      c.gridwidth = 3;
      c.weightx = Columna_2;
      Etiqueta1("Translación", malla, c);
      c.gridwidth = 3;
      c.weightx = Columna_3;
      Etiqueta1("Color", malla, c);
      c.gridheight = 5;
      c.gridwidth = 1;
      c.gridx = 5;
      c.weightx = Columna_3 / 3.0;
      colr = Barra1(malla, c, Scrollbar.VERTICAL, 0, 0, 255);
      c.gridx = 6;
      colg = Barra1(malla, c, Scrollbar.VERTICAL, 0, 0, 255);
      c.gridx = 7;
      colb = Barra1(malla, c, Scrollbar.VERTICAL, 0, 0, 255);
      c.gridwidth = 1;
      c.gridheight = 1;
      c.gridy = 1;
      c.gridx = 2;
      c.weightx = Columna_2 / 3.0;
      transx = Barra1(malla, c,
                  Scrollbar.HORIZONTAL,Rango_trans / 2, 0, Rango_trans);
      c.gridx = 3;
      transy = Barra1(malla, c,
                  Scrollbar.HORIZONTAL,Rango_trans / 2, 0, Rango_trans);
```

Programa 11.4: Inserción y transformación de objetos con **Java** `vrmljava2.java` (Parte **2**/6).

```
      c.gridx = 4;
      transz = Barra1(malla, c,
                  Scrollbar.HORIZONTAL,Rango_trans / 2, 0, Rango_trans);
      c.gridwidth = 2;
      c.gridheight = 2;
      c.weightx = Columna_1;
      c.gridx = 0;
      c.gridy = 2;
      Boton1("Esfera", malla, c);
      c.gridwidth = 3;
      c.gridheight = 1;
      c.gridx = 2;
      c.weightx = Columna_2;
      Etiqueta1("Rotación", malla, c);
      c.gridwidth = 1;
      c.gridx = 2;
      c.gridy = 3;
      c.weightx = Columna_2 / 3.0;
      rotx = Barra1(malla, c, Scrollbar.HORIZONTAL, 0, 0, 360);
      c.gridx = 3;
      roty = Barra1(malla, c, Scrollbar.HORIZONTAL, 0, 0, 360);
      c.gridx = 4;
      rotz = Barra1(malla, c, Scrollbar.HORIZONTAL, 0, 0, 360);
      c.gridwidth = 2;
      c.gridheight = 2;
      c.weightx = Columna_1;
      c.gridx = 0;
      c.gridy = 4;
      Boton1("Cono", malla, c);
      c.gridx = 2;
      c.gridwidth = 3;
      c.gridheight = 1;
      c.weightx = Columna_2;
      Etiqueta1("Escala", malla, c);
      c.gridx = 2;
      c.gridy = 5;
      c.gridheight = 1;
      c.gridwidth = 1;
      c.weightx = Columna_2 / 3.0;
      escx = Barra1(malla, c, Scrollbar.HORIZONTAL,10, 1, 100);
      c.gridx = 3;
      escy = Barra1(malla, c, Scrollbar.HORIZONTAL,10, 1, 100);
      c.gridx = 4;
      escz = Barra1(malla, c, Scrollbar.HORIZONTAL,10, 1, 100);
      }

/* Inicia el applet                                                    */
public void start() {
      System.out.println("Inicio 2...");
      grupo = new Vector();   /* Almacena el mundo                    */
      indice = -1;            /* Contador                             */
      /* Navegador en donde se carga el mundo                          */
      browser = (Browser) vrml.external.Browser.getBrowser(this);
      System.out.println("Navegador: " + browser);
```

Programa 11.4: Inserción y transformación de objetos con **Java** `vrmljava2.java` (Parte **3**/6).

```
      try   {
            /* Mundo VRML en donde cargar los objetos                          */
            root = browser.getNode("ROOT");
            System.out.println("Nodo ROOT " + root);
            }
      catch (InvalidNodeException e) {
            System.out.println("PROBLEMA: " + e);
      }
      System.out.println("Iniciado.");
      }

/* Regresa el navegador a emplear                                             */
public Browser getBrowser() {
      return browser;
      }

/* Maneja los eventos de los botones y Barras                                 */
public boolean handleEvent(Event evento) {
    if (evento.target instanceof Scrollbar){
            if (Grupo_actual == null) return true;
            /* Se activa si alguna barra ha cambiado                          */
            Scrollbar barra = (Scrollbar) evento.target;
            if ((barra == transx) || (barra == transy) || (barra == transz)) {
                  float[] val = new float[3];
                  /* Calcula el nuevo valor del centro al punto               */
                  val[0] = (float) transx.getValue() –
                           (float) Rango_trans / 2.0f;
                  val[1] = (float) transy.getValue() –
                           (float) Rango_trans / 2.0f;
                  val[2] = (float) transz.getValue() –
                           (float) Rango_trans / 2.0f;
                  Grupo_actual.set_translation.setValue(val);
                  }

            if ((barra == escx) || (barra == escy) || (barra == escz)){
                  float[] val = new float[3];
                  /* Calcula el nuevo valor del centro al punto               */
                  val[0] = ((float) escx.getValue()) / 10.0f;
                  val[1] = ((float) escy.getValue()) / 10.0f;
                  val[2] = ((float) escz.getValue()) / 10.0f;
                  Grupo_actual.set_scale.setValue(val);
                  }

            if ((barra == colr) || (barra == colg) || (barra == colb)) {
                  float[] val = new float[3];
                  /* Calcula el nuevo valor del centro al punto               */
                  val[0] = (float) (255 - colr.getValue()) / 255.0f;
                  val[1] = (float) (255 - colg.getValue()) / 255.0f;
                  val[2] = (float) (255 - colb.getValue()) / 255.0f;
                  Grupo_actual.set_diffuseColor.setValue(val);
                  }
            /* Realiza la rotación de acuerdo a la barra                      */
            if ((barra == rotx) || (barra == roty) || (barra == rotz)) {
                  Grupo_actual.xang = rotx.getValue();
                  Grupo_actual.yang = roty.getValue();
                  Grupo_actual.zang = rotz.getValue();
```

Programa 11.4: Inserción y transformación de objetos con **Java** **`vrmljava2.java`** (Parte **4**/6).

```
                V2Rotacion xrot = new V2Rotacion
                      (new V2Vec3D(1.0f, 0.0f, 0.0f),
                      (float) Grupo_actual.xang *
                      (float) (Math.PI / 180.0));
                V2Rotacion yrot = new V2Rotacion
                      (new V2Vec3D(0.0f, 1.0f, 0.0f),
                      (float) Grupo_actual.yang *
                      (float) (Math.PI / 180.0));
                V2Rotacion zrot = new V2Rotacion
                      (new V2Vec3D(0.0f, 0.0f, 1.0f),
                      (float) Grupo_actual.zang *
                      (float) (Math.PI / 180.0));
                V2Rotacion r1 = xrot.producto(yrot);
                V2Rotacion r2 = r1.producto(zrot);
                V2Vec3D eje_1 = new V2Vec3D();
                float angulo = r2.Leer_Valor(eje_1);
                float[] val = new float[4];
                float[] Val_eje = eje_1.Leer_Valor();
                val[0] = Val_eje[0];
                val[1] = Val_eje[1];
                val[2] = Val_eje[2];
                val[3] = angulo;
                Grupo_actual.set_rotation.setValue(val);
                }
          return true;
         }
      return super.handleEvent(evento);
      }

/* Maneja el accionamiento de los botones                                  */
public boolean action(Event evento, Object what) {
    if (evento.target instanceof Button) {
          V2Grupo Grupo_nuevo = null;   /* Grupo de objetos                */
          /* Agrega objeto según el botón                                  */
          Button b = (Button) evento.target;
          if (b.getLabel() == "Cubo")
                Grupo_nuevo = new V2Grupo(this, V2Grupo.CUBE);
          if (b.getLabel() == "Esfera")
                Grupo_nuevo = new V2Grupo(this, V2Grupo.SPHERE);
          if (b.getLabel() == "Cono")
                Grupo_nuevo = new V2Grupo(this, V2Grupo.CONE);
          grupo.addElement(Grupo_nuevo);
          indice++;
          Grupo_actual = Grupo_nuevo;
          Inicia_Grupo();
          try {
                EventInMFNode addChildren = (EventInMFNode)
                            root.getEventIn("addChildren");
                addChildren.setValue(Grupo_actual.Arreglo_Trans);
                }
          catch (InvalidEventInException e) {
                System.out.println("PROBLEMA 1: " + e);
                }
          }
      return true;
      }
```

Programa 11.4: Inserción y transformación de objetos con **Java** `vrmljava2.java` (Parte **5**/6).

```
/* Guarda grupo actual e inicia el nuevo grupo de objetos                    */
public void Actual(V2Grupo a) {
    Grupo_actual = a;
    Inicia_Grupo();
      }

/* Inicia el nuevo grupo de objetos                                          */
void Inicia_Grupo() {
    float[] val = Grupo_actual.scale_changed.getValue();
    escx.setValue((int) (val[0] * 10.0f));
    escy.setValue((int) (val[1] * 10.0f));
    escz.setValue((int) (val[2] * 10.0f));

    val = Grupo_actual.translation_changed.getValue();
    transx.setValue((int) val[0] + (Rango_trans / 2));
    transy.setValue((int) val[1] + (Rango_trans / 2));
    transz.setValue((int) val[2] + (Rango_trans / 2));

    val = Grupo_actual.diffuseColor_changed.getValue();
    colr.setValue(255 - (int) (val[0] * 255.0f));
    colg.setValue(255 - (int) (val[1] * 255.0f));
    colb.setValue(255 - (int) (val[2] * 255.0f));

    rotx.setValue(Grupo_actual.xang);
    roty.setValue(Grupo_actual.yang);
    rotz.setValue(Grupo_actual.zang);
      }
}
```

Programa 11.4: Inserción y transformación de objetos con **Java** `vrmljava2.java` (Parte **6**/6).

```
/* Encapsula Objetos para presentarlos en la escena virtual                  */
import java.lang.*;          /* Paquete de Java                              */
import java.applet.*;        /* Clase Applet                                 */
/* Paquetes de VRML para interactuar con Java                                */
import vrml.external.field.*;
import vrml.external.exception.*;
import vrml.external.Node;
import vrml.external.Browser;
/* Programa principal                                                        */
import vrmljava2;
public class V2Grupo extends Object implements EventOutObserver {
      public final static int CUBE = 0;
      public final static int SPHERE = 1;
      public final static int CONE = 2;
      vrmljava2 principal;
      Browser browser;
      /* Declaración de nodos                                                */
      public Node[] Arreglo_Trans;
      Node transform;
      Node material;
      /* Eventos de entrada a modificar                                      */
      public EventInSFRotation set_rotation;
      public EventInSFVec3f set_scale;
      public EventInSFVec3f set_translation;
      public EventInSFColor set_diffuseColor;
```

Programa 11.5: Inserción y transformación de objetos con **Java** `V2Grupo.java` (Parte **1**/3).

```
/* Eventos de salida a modificar                                    */
public EventOutSFRotation rotation_changed;
public EventOutSFVec3f scale_changed;
public EventOutSFVec3f translation_changed;
public EventOutSFColor diffuseColor_changed;
public EventOutSFTime touchTime_changed;
/* Otras variables                                                  */
int xang;
int yang;
int zang;
V2Grupo(vrmljava2 temp_1, int tipo) throws
            IllegalArgumentException {
      principal = temp_1;
      browser = principal.getBrowser();
      xang = yang = zang = 0;
      if ((tipo != V2Grupo.CUBE) && (tipo != V2Grupo.SPHERE)
            && (tipo != V2Grupo.CONE)) {
            throw(new IllegalArgumentException());
            }
      try   {
            /* Nodos comunes                                        */
            Arreglo_Trans =
                  browser.createVrmlFromString("Transform {}");
            Node[] shape_A =
                  browser.createVrmlFromString("Shape {}");
            Node[] mat_A =
                  browser.createVrmlFromString("Material {}");
            Node[] app_A =
                  browser.createVrmlFromString("Appearance {}");
            Node[] sens_A =
                  browser.createVrmlFromString("TouchSensor {}");
            Node[] geom_A =   null;
            if (tipo == V2Grupo.CUBE)
                  geom_A = browser.createVrmlFromString("Box {}");
            if (tipo == V2Grupo.SPHERE)
                  geom_A =
                        browser.createVrmlFromString("Sphere {}");
            if (tipo == V2Grupo.CONE)
                  geom_A =
                        browser.createVrmlFromString("Cone {}");
            transform = Arreglo_Trans[0];
            material = mat_A[0];
            EventInSFNode nodeIn = (EventInSFNode)
                  shape_A[0].getEventIn("appearance");
            nodeIn.setValue(app_A[0]);
            nodeIn = (EventInSFNode)
                  app_A[0].getEventIn("material");
            nodeIn.setValue(material);
            nodeIn = (EventInSFNode)
                  shape_A[0].getEventIn("geometry");
            nodeIn.setValue(geom_A[0]);
            EventInMFNode nodesIn = (EventInMFNode)
                  transform.getEventIn("addChildren");
            nodesIn.setValue(shape_A);
            nodesIn.setValue(sens_A);
            set_rotation = (EventInSFRotation)
                  transform.getEventIn("rotation");
```

Programa 11.5: Inserción y transformación de objetos con **Java** `V2Grupo.java` (Parte **2**/3).

```
                set_scale = (EventInSFVec3f)
                      transform.getEventIn("scale");
                set_translation = (EventInSFVec3f)
                      transform.getEventIn("translation");
                set_diffuseColor = (EventInSFColor)
                      material.getEventIn("diffuseColor");
                rotation_changed = (EventOutSFRotation)
                      transform.getEventOut("rotation");
                scale_changed = (EventOutSFVec3f)
                      transform.getEventOut("scale");
                translation_changed = (EventOutSFVec3f)
                      transform.getEventOut("translation");
                diffuseColor_changed = (EventOutSFColor)
                      material.getEventOut("diffuseColor");
                touchTime_changed = (EventOutSFTime)
                      sens_A[0].getEventOut("touchTime");
                touchTime_changed.advise(this, null);
                }
          catch (InvalidVrmlException e) {
                System.out.println("PROBLEMA 1: " + e);
                }
          catch (InvalidEventInException e) {
                System.out.println("PROBLEMA 2: " + e);
                }
          catch (InvalidEventOutException e) {
          System.out.println("PROBLEMA 3: " + e);
                }
          }
     public void callback(EventOut event,double time,Object userData) {
          principal.Actual(this);
          }
     }
```

Programa 11.5: Inserción y transformación de objetos con **Java** `V2Grupo.java` (Parte **3**/3).

```
/* Implementación de VRML y Java                                        */
import java.lang.*;     /* Paquete del lenguaje Java                    */
import java.util.*;     /* Clases e interfaces de utilidad              */

/* Manipulación de Puntos                                               */
import V2Vec3D;

/* Rota objetos en el espacio                                           */
public class V2Rotacion {
     /* Punto como arreglo                                              */
     V2Rotacion() {
          punto = new float[4];
          }

     /* Carga el arreglo de puntos                                      */
     V2Rotacion(float q0, float q1, float q2, float q3) {
          punto = new float[4];
          punto[0] = q0; punto[1] = q1; punto[2] = q2; punto[3] = q3;
          normalizar();
          }
```

Programa 11.6: Inserción y transformación de objetos con **Java** `V2Rotacion.java` (Parte **1**/2).

```
/* Rota sobre un eje un ángulo en radianes                              */
V2Rotacion(V2Vec3D eje, float radianes) {
      punto = new float[4];
      Valor_1(eje, radianes);
      }

/* Rota sobre un eje un ángulo en radianes                              */
public V2Rotacion Valor_1(V2Vec3D eje, float radianes) {
      V2Vec3D q = new V2Vec3D(eje);
      q.normalizar();
      q.producto((float) Math.sin(radianes / 2.0f));
      float[] val = q.Leer_Valor();
      punto[0] = val[0];
      punto[1] = val[1];
      punto[2] = val[2];
      punto[3] = (float) Math.cos(radianes / 2.0f);
      return this;
      }

/* Lee el valor del punto del objeto a rotar                            */
public float Leer_Valor(V2Vec3D eje) {
      float len;
      V2Vec3D q = new V2Vec3D(punto[0], punto[1], punto[2]);
      if ((len = q.tam_1()) > 0.00001f) {
            q.producto(1.0f / len);
            float[] val = q.Leer_Valor();
            eje.Valor_1(val[0], val[1], val[2]);
            return (float)(2.0 * Math.acos(punto[3]));
            }
      else  {
            eje.Valor_1(0.0f, 0.0f, 1.0f);
            return 0.0f;
            }
      }

/* Calcula la rotación                                                  */
V2Rotacion producto(V2Rotacion q2) {
V2Rotacion  q = new V2Rotacion
      (q2.punto[3] * punto[0] + q2.punto[0] * punto[3] +
      q2.punto[1] * punto[2] - q2.punto[2] * punto[1],
      q2.punto[3] * punto[1] + q2.punto[1] * punto[3] +
      q2.punto[2] * punto[0] - q2.punto[0] * punto[2],
      q2.punto[3] * punto[2] + q2.punto[2] * punto[3] +
      q2.punto[0] * punto[1] - q2.punto[1] * punto[0],
      q2.punto[3] * punto[3] - q2.punto[0] * punto[0] -
      q2.punto[1] * punto[1] - q2.punto[2] * punto[2]);
      q.normalizar();
      return (q);
      }

/* Normaliza puntos                                                     */
void normalizar() {
      float dist = (float)(1.0 / Math.sqrt(norm()));
      punto[0] *= dist;
      punto[1] *= dist;
      punto[2] *= dist;
```

Programa 11.6: Inserción y transformación de objetos con **Java** `V2Rotacion.java` (Parte **2**/3).

```
            punto[3] *= dist;
            }

      /* Otra parte de la normalización                                    */
      float norm() {
      return (punto[0] * punto[0] +
            punto[1] * punto[1] +
            punto[2] * punto[2] +
            punto[3] * punto[3]);
            }
      float[] punto;
      }
```

Programa 11.6: Inserción y transformación de objetos con **Java** `V2Rotacion.java` (Parte **3**/3).

```
/* Rutina de manejo de vectores para  la implementación de VRML y Java    */
import java.lang.*;     /* Clase del lenguaje Java                        */

/* Calcula la posición de un vector en el espacio                         */
public class V2Vec3D {

      /* Nuevo vector                                                      */
      V2Vec3D() {
            vec = new float[3];
            }

      /*Carga el vector                                                   */
      V2Vec3D(V2Vec3D arg) {
            vec = new float[3];
            vec[0] = arg.vec[0];
            vec[1] = arg.vec[1];
            vec[2] = arg.vec[2];
            }

      /* Obtiene valores para el vector                                   */
      V2Vec3D(float x, float y, float z) {
            vec = new float[3];
            Valor_1(x, y, z);
            }

      /* Distancia del centro al punto                                    */
      public float tam_1() {
      return (float) Math.sqrt(vec[0] * vec[0] +
            vec[1] * vec[1] +
            vec[2] * vec[2]);
            }

      /* Normaliza vector                                                 */
      public void normalizar() {
            float tam = tam_1();
            if (tam != 0.0) producto(tam);
            else Valor_1(0.0f, 0.0f, 0.0f);
            }
```

Programa 11.7: Inserción y transformación de objetos con **Java** `V2Vect3D.java` (Parte **1**/2).

```
      /* Carga el valor del vector                                          */
      public void Valor_1(float x, float y, float z) {
            vec[0] = x; vec[1] = y; vec[2] = z;
            }

      /* Lee el vector                                                      */
      float[] Leer_Valor() {
            return vec;
            }

      /* Realiza el producto para recuperar la posición                     */
      void producto(float arg) {
            vec[0] *= arg;
            vec[1] *= arg;
            vec[2] *= arg;
            }

      float[] vec;
      }
```

Programa 11.7: Inserción y transformación de objetos con **Java** `V2Vect3D.java` (Parte **2**/2).

En el **Programa 11.8** se tiene el código **HTML**, necesario para cargar el **applet** de **Java** y el mundo **VRML**; en el **Programa 11.9** se tiene un mundo vacío.

```
<html>

<head>
<title>Ejemplo 10.2: Uso de VRML con Java</title>
</head>

<body bgcolor="#000000" text="#FFFFFF">

<p align="center">
<embed src="root.wrl" border="0" height="300" width="500">
<applet code="vrmljava2.class" width="500" height="120" mayscript>
</applet>
</p>

</body>
</html>
```

Programa 11.8: Código **HTML** para mostrar un mundo **VRML** y ejecutar un **applet** de **Java**.

```
#VRML V2.0 utf8
NavigationInfo { type "EXAMINE" }

DEF ROOT Group {}
```

Programa 11.9: Programa `root.wrl` del mundo virtual vacío.

El resultado se muestra en la **Figura 11.2**.

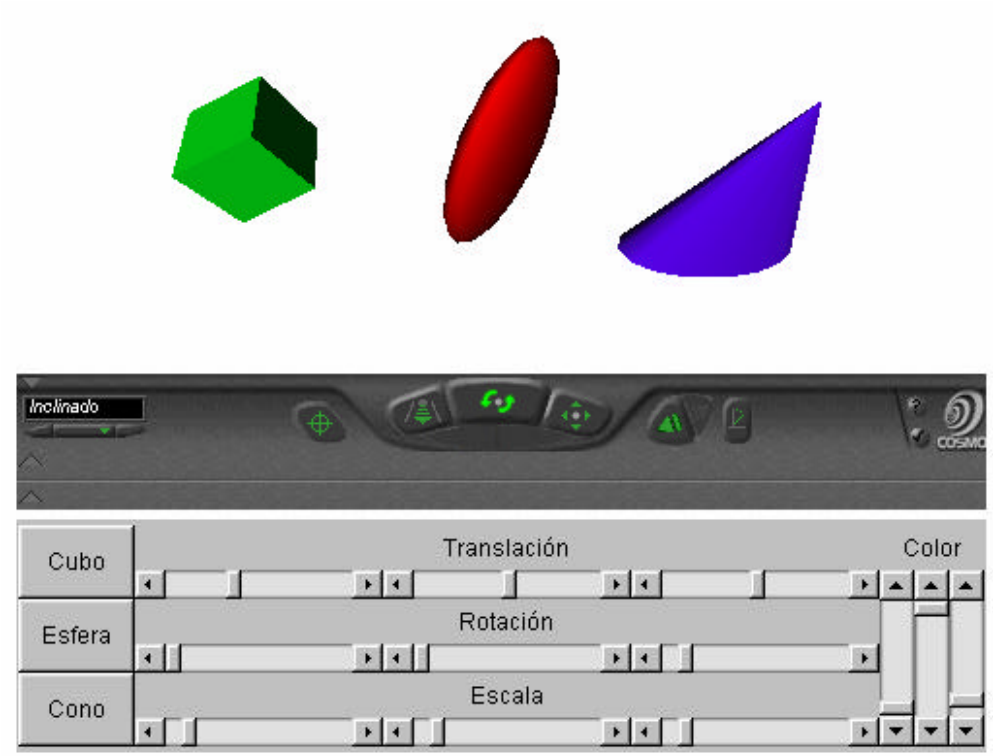


Figura 11.2: Interacción de **VRML** y **Java** con diversos objetos.

Capítulo 12: Introducción a X3D

X3D es un lenguaje de programación para gráficos vectoriales definido por una norma **ISO** [X3D 04], que puede emplear tanto una sintaxis similar a la de **XML** como una del tipo de **VRML**. **X3D** amplía **VRML** con extensiones de diseño y la posibilidad de emplear **XML** para modelar escenas completas en tiempo real.

12.1 Creación de mundos X3D

Los visores para **X3D** pueden ser los mismos que los usados para **VRML**, de hecho un mundo virtual en **VRML**, puede ser insertado en un mundo **X3D**; la lista de visores para **X3D** para cada plataforma se muestra a continuación:

Para **Windows**:

- BS Contact: `www.bitmanagement.com/`
- Octaga Player: `www.octaga.com/`

Para **UNIX/Linux**:

- Octaga Player: `www.octaga.com/`
- Xj3D: `www.xj3d.org/`

Para **MacOS**:

- MacWeb3D: `www.macweb3d.org`
- Xj3D: `www.xj3d.org/`

Para crear mundos **X3D** se tienen dos opciones:

- **Codificarlos a mano**, y para ello solo se necesita un editor del texto, como el **notepad** o el **wordpad**. Una vez realizado el código, se debe salvar como archivo `*.x3d`; pero debemos asegurarnos de guardar el archivo como **texto sin formato**, al igual que con los archivos `*.html`.
- Se puede usar una de las muchas **herramientas de autoría** de **X3D** o **modeladores de X3D**.

12.2 Formato de archivos

Los archivos de **X3D**, son archivos de texto (**ASCII**), por lo que pueden ser modificados en un editor de texto convencional. Un archivo **X3D** tiene la extensión `*.x3d`.

La primera línea en un archivo **X3D** es con la que el navegador identifica la versión del archivo, es decir, la **cabecera**. Para **X3D** la cabecera es la siguiente: `<?xml version="1.0" encoding="UTF-8"?>`. Donde `xml version="1.0"` denota el tipo y la versión, y `encoding="UTF-8"` permite utilizar la codificación **UTF-8** para poder emplear todos los caracteres especiales del estándar **ISO 10646**.

Los comentarios en **X3D** se denotan colocándolos de la forma: `<--Comentario-->`

X3D es **sensible al contexto**, es decir, una variable en minúsculas es diferente a la misma variable en mayúsculas.

12.3 Primitivas y materiales

Existen en **X3D** algunas primitivas que definen una serie de objetos simples. Para poder visualizarlas, hay que usar un nodo `Shape`, el cual presenta dos campos `geometry` y `appearance`. Las primitivas para el campo `geometry` son:

- **Box size="x y z"** para dibujar un **cubo** de dimensiones **x**, **y** y **z** para cada uno de los ejes coordenados.
- **Sphere="radius r"** para dibujar una **esfera** con radio **r**.
- **Cone="bottomRadius r height h"** para dibujar un **cono** con una circunferencia en la base de radio **r** y altura **h**.
- **Cylinder="radius r height h"** para dibujar un **cilindro** con una circunferencia de radio **r** y altura **h**.

Las primitivas **X3D** anteriores definen la geometría de los objetos básicos. También es posible definir un **color** ó alguna **textura**. La **apariencia** de una primitiva se establece por medio del campo **Appearance** del nodo **Shape**. Algunas primitivas de este nodo son:

- **Material**.- para definir un material de la primitiva como **color** y **textura**.
- **diffuseColor R G B**.- para establecer el color del material empleado. Se emplea el formato de color **RGB** (**R** para el **rojo**, **G** para el **verde** y **B** para el **azul**). En las **Tablas 12.1** y **12.2** es posible observar como se combinan estos parámetros para proporcionar múltiples colores y texturas.

Color	Color RGB
Blanco	1 1 1
Amarillo	1 1 0
Cyan	0 1 1
Verde	0 1 0
Magenta	1 0 1
Rojo	1 0 0
Azul	0 0 1

Tabla 12.1: Colores básicos en **X3D**.

Descripción	AmbientColor	DiffuseColor	SpecularColor	shininess
Dorado	0.57 0.40 0.00	0.22 0.15 0.00	0.71 0.70 0.56	0.16
Aluminio	0.30 0.30 0.35	0.30 0.30 0.50	0.70 0.70 0.80	0.09
Cobre	0.33 0.26 0.23	0.50 0.11 0.00	0.95 0.73 0.00	0.93
Púrpura	0.25 0.17 0.19	0.10 0.03 0.22	0.64 0.00 0.98	0.08

Tabla 12.2: Colores metálicos con otros campos de **X3D**.

El **Programa 12.1** muestra un ejemplo sencillo de un programa en **X3D**. Este programa dibuja una caja amarilla con dimensiones en **x** de **4** unidades, en **y** de **3** unidades y en **z** con **5** unidades (**Figura 12.1**).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "www.geocities.com/daraujo24">
<X3D profile="Full">
  <head>
    <meta name="Nombre" content="p_12_01.x3d"/>
  </head>
  <Scene>
  <Shape>
      <Appearance >
          <Material  diffuseColor="1.0 1.0 0.0"/>
      </Appearance>
      <Box  size="4.0 3.0 5.0"/>
  </Shape>
  </Scene>
</X3D>
```

Programa 12.1: Uso de las primitivas básicas de **X3D**.

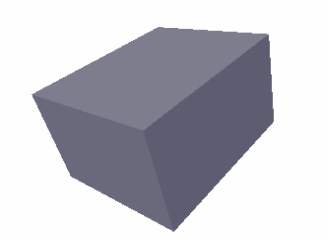


Figura 12.1: Caja simple en **X3D**.

12.4 Texto en X3D

X3D provee una forma de agregar texto como si se tratase de objetos, a través de la primitiva **Text** del campo **geometry**, en donde también es posible definir otros parámetros de la letra como son el tipo, estilo, tamaño, etc. (**Programa 12.2** y **Figura 12.2**).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "www.geocities.com/daraujo24">
<X3D profile="Full">
  <head>
    <meta name="Nombre" content="p_12_02.x3d"/>
   </head>
  <Scene>
  <Shape >
      <Appearance >
          <Material     ambientIntensity="0.143"
                        emissiveColor="0.09 0.04 0.03"
                        shininess="0.13"
                        diffuseColor="0.91 0.44 0.35"
                        specularColor="0.35 0.14 0.0"/>
      </Appearance>
      <Text  string='"X3D"'>
          <FontStyle  family='"Verdana"' size="1.5"/>
      </Text>
  </Shape>
  </Scene>
</X3D>
```

Programa 12.2: Código **X3D** para insertar texto.

Figura 12.2: Texto en el espacio virtual con **X3D**.

12.5 Información del mundo

Es la información que el navegador presenta al usuario, como es el **título del mundo** (`title`) y otro tipo de información como: autores, palabras clave, etc. (`info`), por ejemplo tenemos el **Programa 12.3**, que simplemente agrega información al mundo.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "www.geocities.com/daraujo24">
<X3D profile="Full">
  <head>
    <meta name="Nombre" content="p_12_03.x3d"/>
  </head>
  <Scene>
  <WorldInfo  info='"Programa 12.3
                David Araujo Díaz
                www.geocities.com/daraujo24
                "' title="Título que se presenta "/>
  </Scene>
</X3D>
```

Programa 12.3: Información del mundo virtual.

12.6 Fondo

Para colocar un color de fondo como sólido ó color degradado, **X3D** emplea el concepto de esfera de radio infinito que engloba a todo el mundo. Para la manipulación de la esfera se emplea el nodo `Background`.

Para definir **un solo color del cielo**, se hace empleando el campo `skyColor` de la forma en que se muestra en el **Programa 12.4** (**Figura 12.3**).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "www.geocities.com/daraujo24">
<X3D profile="Full">
  <head>
    <meta name="Nombre" content="p_12_04.x3d"/>
  </head>
  <Scene>
  <Background  skyColor="0.8 0.8 1.0 "/>
  <Shape >
      <Appearance >
          <Material  diffuseColor="0.0 0.82 0.98"/>
      </Appearance>
      <Sphere  radius="2.0"/>
  </Shape>
  </Scene>
</X3D>
```

Programa 12.4: Definir un solo color del cielo.

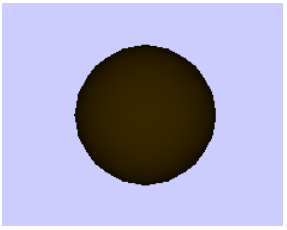


Figura 12.3: Un solo color del cielo como fondo.

Para definir un fondo con **color de cielo degradado** (**Programa 12.5** y **Figura 12.4**).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "www.geocities.com/daraujo24">
<X3D profile="Full">
  <head>
    <meta name="Nombre" content="p_12_05.x3d"/>
  </head>
  <Scene>
  <Background     skyAngle="  0.384 0.785 1.047 1.309 1.484 1.5708"
                  skyColor="  0.0 0.0 0.2, 0.0 0.0 1.0, 0.0 1.0 1.0, 0.75 0.75 1.0,
                              0.8 0.8 0.0, 0.8 0.6 0.0, 1.0 0.4 0.0, "/>
  <Shape >
      <Appearance >
          <Material  diffuseColor="0.0 0.82 0.98"/>
      </Appearance>
      <Sphere  radius="2.0"/>
  </Shape>
  </Scene>
</X3D>
```

Programa 12.5: Definir colores del cielo degradados.

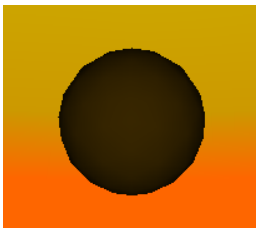


Figura 12.4: Cielo con colores degradados como fondo.

Para definir un fondo con **color de cielo degradado** y **color de tierra**, se muestra el **Programa 12.6** y la **Figura 12.5**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "www.geocities.com/daraujo24">
<X3D profile="Full">
  <head>
     <meta name="Nombre" content="p_12_06.x3d"/>
  </head>
  <Scene>
  <Background     groundAngle=" 1.5708"
                  groundColor=" 0.1 0.8 0.2, 0.1 0.8 0.2,"
                  skyAngle="  0.384 0.785 1.047 1.309 1.484 1.5708"
                  skyColor="  0.0 0.0 0.2, 0.0 0.0 1.0, 0.0 1.0 1.0, 0.75 0.75 1.0,
                              0.8 0.8 0.0, 0.8 0.6 0.0, 1.0 0.4 0.0, "/>
  <Shape >
      <Appearance >
          <Material  diffuseColor="0.0 0.82 0.98"/>
      </Appearance>
      <Sphere  radius="2.0"/>
  </Shape>
  </Scene>
</X3D>
```

Programa 12.6: Definir colores del cielo degradados y color de tierra.

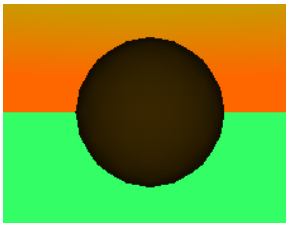


Figura 12.5: Colores del cielo degradados y color de tierra como fondo.

Para definir un **fondo con seis imágenes alrededor** se tiene el **Programa 12.7** y la **Figura 1.6**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "www.geocities.com/daraujo24">
<X3D profile="Full">
  <head>
    <meta name="Nombre" content="p_12_07.x3d"/>
  </head>
  <Scene>
  <Background     bottomUrl='"tierra.jpg"'
                  backUrl='"atras.jpg"'
                  leftUrl='"izquierda.jpg"'
                  topUrl='"arriba.jpg"'
                  frontUrl='"frente.jpg"'
                  rightUrl='"derecha.jpg"'/>
  <Shape >
      <Appearance >
          <Material  diffuseColor="0.0 0.82 0.98"/>
      </Appearance>
      <Sphere  radius="2.0"/>
  </Shape>
  </Scene>
</X3D>
```

Programa 12.7: Fondo con seis imágenes alrededor.

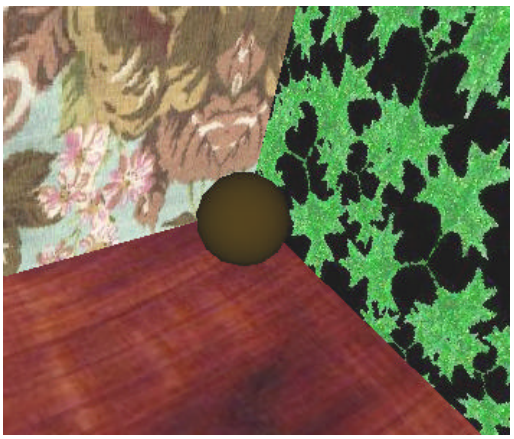


Figura 12.6: Fondo con seis imágenes alrededor.

12.7 Animación con X3D

Los sensores se emplean principalmente para **animar** algunos objetos dentro de un mundo virtual. En el **Programa 12.8** se observa el uso de los sensores de tiempo y de coordenadas para simular el movimiento de una lombriz (**Figura 12.7**).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "www.geocities.com/daraujo24">
<X3D profile="Full">
  <head>
    <meta name="filename" content="p_12_08.x3d"/>
  </head>
  <Scene>
  <Transform  translation="0.0 0.3 0.0">
      <Shape >
          <Appearance >
              <Material  diffuseColor="0.0 1.0 0.2"/>
          </Appearance>
          <Extrusion  DEF="Cuerpo"
                  crossSection="1.0 0.0, 0.92 -0.38, 0.71 -0.71, 0.38 -0.92, 0.0 -
1.0, -0.38 -0.92, -0.71 -0.71, -0.92 -0.38, -1.0 -0.0, -0.92 0.38, -0.71 0.71, -
0.38 0.92, 0.0 1.0, 0.38 0.92, 0.71 0.71, 0.92 0.38, 1.0 0.0, "
                  scale="0.05 0.02, 0.2 0.1, 0.4 0.15, 0.3 0.3, 0.3 0.3, 0.3 0.3,
0.3 0.3, 0.3 0.3, 0.3 0.3, 0.3 0.3, 0.29 0.29, 0.29 0.29, 0.29 0.29, 0.28 0.28,
0.28 0.28, 0.25 0.25, 0.2 0.2, 0.1 0.1, 0.05 0.05, "
                  creaseAngle="1.57"
                  spine="-4.1 0.0 0.0,  -4.0 0.0 0.0,  -3.529 0.0 0.674,  -3.059 0.0
0.996,  -2.588 0.0 0.798,  -2.118 0.0 0.184,  -1.647 0.0 -0.526,  -1.176 0.0 -0.962,  -
0.706 0.0 -0.895,  -0.235 0.0 -0.361,  0.235 0.0 0.361,  0.706 0.0 0.895,  1.176 0.0
0.962,  1.647 0.0 0.526,  2.118 0.0 -0.184,  2.588 0.0 -0.798,  3.059 0.0 -0.996,  3.529
0.0 -0.674,  4.0 0.0 0.0, "/>
      </Shape>
  </Transform>
  <TimeSensor  DEF="Tiempo"  loop="true" cycleInterval="4.0"/>
  <CoordinateInterpolator  DEF="Gusano"
                  key=" 0.0 0.25 0.5 0.75 1.0"
                  keyValue="-4.1 0.0 0.0,  -4.0 0.0 0.0,  -3.529 0.0 0.674,  -3.059
0.0 0.996,  -2.588 0.0 0.798,  -2.118 0.0 0.184,  -1.647 0.0 -0.526,  -1.176 0.0 -
0.962,  -0.706 0.0 -0.895,  -0.235 0.0 -0.361,  0.235 0.0 0.361,  0.706 0.0 0.895,
1.176 0.0 0.962,  1.647 0.0 0.526,  2.118 0.0 -0.184,  2.588 0.0 -0.798,  3.059 0.0 -
0.996,  3.529 0.0 -0.674,  4.0 0.0 0.0,  -4.1 0.0 -1.0,  -4.0 0.0 -1.0,  -3.529 0.0 -
0.739,  -3.059 0.0 -0.092,  -2.588 0.0 0.603,  -2.118 0.0 0.983,  -1.647 0.0 0.85,  -
1.176 0.0 0.274,  -0.706 0.0 -0.446,  -0.235 0.0 -0.932,  0.235 0.0 -0.932,  0.706 0.0
-0.446,  1.176 0.0 0.274,  1.647 0.0 0.85,  2.118 0.0 0.983,  2.588 0.0 0.603,  3.059
0.0 -0.092,  3.529 0.0 -0.739,  4.0 0.0 -1.0,  -4.1 0.0 0.0,  -4.0 0.0 0.0,  -3.529 0.0
-0.674,  -3.059 0.0 -0.996,  -2.588 0.0 -0.798,  -2.118 0.0 -0.184,  -1.647 0.0 0.526,
-1.176 0.0 0.962,  -0.706 0.0 0.895,  -0.235 0.0 0.361,  0.235 0.0 -0.361,  0.706 0.0 -
0.895,  1.176 0.0 -0.962,  1.647 0.0 -0.526,  2.118 0.0 0.184,  2.588 0.0 0.798,  3.059
0.0 0.996,  3.529 0.0 0.674,  4.0 0.0 0.0,  -4.1 0.0 1.0,  -4.0 0.0 1.0,  -3.529 0.0
0.739,  -3.059 0.0 0.092,  -2.588 0.0 -0.603,  -2.118 0.0 -0.983,  -1.647 0.0 -0.85,  -
1.176 0.0 -0.274,  -0.706 0.0 0.446,  -0.235 0.0 0.932,  0.235 0.0 0.932,  0.706 0.0
0.446,  1.176 0.0 -0.274,  1.647 0.0 -0.85,  2.118 0.0 -0.983,  2.588 0.0 -0.603,  3.059
0.0 0.092,  3.529 0.0 0.739,  4.0 0.0 1.0,  -4.1 0.0 0.0,  -4.0 0.0 0.0,  -3.529 0.0
0.674,  -3.059 0.0 0.996,  -2.588 0.0 0.798,  -2.118 0.0 0.184,  -1.647 0.0 -0.526,  -
1.176 0.0 -0.962,  -0.706 0.0 -0.895,  -0.235 0.0 -0.361,  0.235 0.0 0.361,  0.706 0.0
0.895,  1.176 0.0 0.962,  1.647 0.0 0.526,  2.118 0.0 -0.184,  2.588 0.0 -0.798,  3.059
0.0 -0.996,  3.529 0.0 -0.674,  4.0 0.0 0.0, "/>
```

Programa 12.8: **Animación** con **X3D** (Parte **1**/2).

```
<ROUTE      fromNode="Tiempo"
            fromField="fraction_changed"
            toNode="Gusano"
            toField="set_fraction"/>
<ROUTE      fromNode="Gusano"
            fromField="value_changed"
            toNode="Cuerpo"
            toField="set_spine"/>
  </Scene>
</X3D>
```

Programa 12.8: **Animación** con **X3D** (Parte **2**/2).

Figura 12.7: **Animación** con sensores.

Apéndice A: Transformaciones Geométricas

La representación de objetos en la computadora requiere de la manipulación matemática de cada uno de los **puntos** (ó píxeles) que componen un objeto en la pantalla. Está representación enriquece y facilita el trabajo de los usuarios.

Cada objeto es representado internamente como un **conjunto de puntos** que guardan alguna relación entre sí, para facilitar su manejo y las operaciones entre ellos. En este apartado se presentan algunas operaciones con objetos reconocidos o modelados para su representación en forma gráfica en dos o tres dimensiones.

Se presentan las transformaciones geométricas bidimensionales y tridimensionales, necesarias para representar información en forma gráfica en la pantalla de la computadora. Las transformaciones de traslación, escala y rotación son fundamentales para trabajar con operaciones gráficas en espacios de dos y tres dimensiones.

a.1 Transformaciones geométricas bidimensionales

Para un **punto** (ó píxel en el caso de imágenes) en el plano **P(x,y)** podemos realizar las operaciones siguientes en el **espacio bidimensional [Foley 96]**.

a.1.1 Traslación

Para **trasladar** un objeto, se añaden los valores de las coordenadas de traslación, a cada uno de los puntos que lo conforman (**Ecs. a.1** y **a.2**).

$$x' = x + d_x \quad \text{...(Ec. a.1)}$$
$$y' = y + d_y \quad \text{...(Ec. a.2)}$$

Donde: **x**, **y** = Punto de entrada.
d_x = Desplazamiento en **x**.
d_y = Desplazamiento en **y**.

a.1.2 Escala

La **escala** permite cambiar el tamaño de los objetos. El escalamiento se lleva a cabo respecto al origen. Es **diferencial** si el escalamiento es distinto para ambos **ejes (x,y)** y **uniforme** si es el mismo (**Ecs. a.3** y **a.4**).

$$x' = s_x * x \quad \text{...(Ec. a.3)}$$
$$y' = s_y * y \quad \text{...(Ec. a.4)}$$

Donde: **x**, **y** = Punto de entrada.
s_x = Escala en ***x***.
s_y = Escala en **y**.

a.1.3 Rotación

La **rotación** puede hacer que un objeto gire en un **ángulo** α (**Ecs. a.5** y **a.6**).

$$y' = x * \cos\alpha - y * \sin\alpha \quad \text{...(Ec. a.5)}$$
$$y' = x * \sin\alpha + y * \cos\alpha \quad \text{...(Ec. a.6)}$$

Donde: **x**, **y** = Punto de entrada.
α = Angulo de rotación.

a.1.4 Sesgo

El **sesgo** es de dos tipos: sesgo sobre el **eje y** y sesgo sobre el **eje x**. En el **sesgo** las líneas paralelas permanecen paralelas (**Ecs. a.7**, **a.8**, **a.9** y **a.10**).

Sesgo sobre el **eje X**:

$$x' = x + a * y \quad \text{...(Ec. a.7)}$$
$$y' = y \quad \text{...(Ec. a.8)}$$

Sesgo sobre el **eje Y**:

$$x' = x \quad \text{...(Ec. a.9)}$$
$$y' = b * x + y \quad \text{...(Ec. a.10)}$$

Donde: **x**, **y** = Punto de entrada.
a, **b** = Constantes de proporcionalidad.

a.1.5 Representación de objetos bidimensionales

Para poder realizar las operaciones con mayor facilidad, se representa cada punto por sus **coordenadas (x,y)** y las rectas que los unen, numerando los puntos y uniéndolos mediante sus índices como se muestra en la **Figura a.1**.

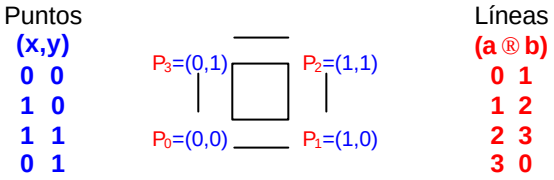
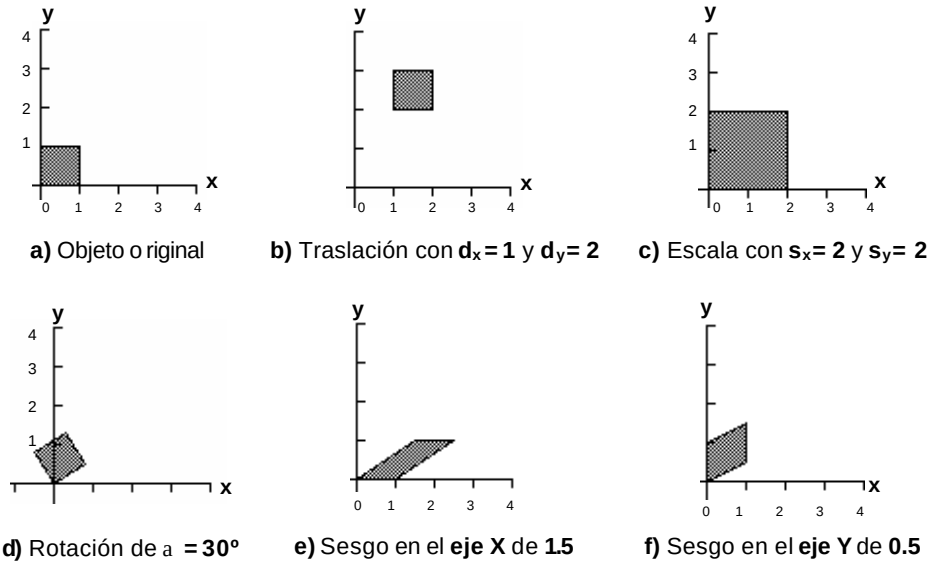


Figura a.1: Representación de objetos bidimensionales.

a.1.6 Operaciones con transformaciones bidimensionales

En la **Figura a.2** se observan los resultados de aplicar las transformaciones bidimensionales a un cuadrado.



a) Objeto original **b)** Traslación con $d_x = 1$ y $d_y = 2$ **c)** Escala con $s_x = 2$ y $s_y = 2$

d) Rotación de a **= 30°** **e)** Sesgo en el **eje X** de **1.5** **f)** Sesgo en el **eje Y** de **0.5**

Figura a.2: Resultados de las **transformaciones bidimensionales** en un **cuadrado**.

a.2 Transformaciones geométricas tridimensionales

Para un **punto** (ó píxel en el caso de imágenes) en el espacio **P(x,y,z)** podemos realizar las operaciones siguientes en el **espacio tridimensional [Foley 96]**.

a.2.1 Traslación

La **traslación** se usa para mover un objeto en el espacio. Es necesario añadir las coordenadas en cada uno de los tres ejes (**Ecs. a.11**, **a.12** y **a.13**).

$$x' = x + d_x \quad \text{...(Ec. a.11)}$$
$$y' = y + d_y \quad \text{...(Ec. a.12)}$$
$$z' = z + d_z \quad \text{...(Ec. a.13)}$$

Donde: **x**, **y**, **z** = Punto de entrada.
d_x = Desplazamiento en **x**.
d_y = Desplazamiento en **y**.
d_z = Desplazamiento en **z**.

a.2.2 Escala

La **escala** cambia el tamaño en alguna de las dimensiones de un objeto en el espacio. Es necesario aplicar la **escala** a cada una de las coordenadas de sus puntos (**Ecs. a.14**, **a.15** y **a.16**).

$$x' = s_x * x \quad \text{...(Ec. a.14)}$$
$$y' = s_y * y \quad \text{...(Ec. a.15)}$$
$$z' = s_z * z \quad \text{...(Ec. a.16)}$$

Donde: **x**, **y**, **z** = Punto de entrada.
s_x = Escala en **x**.
s_y = Escala en **y**.
s_z = Escala en **z**.

a.2.3 Rotación

En el espacio tridimensional, existen tres **rotaciones** una por cada eje (**Ecs. a.17**, **a.18**, **a.19**, **a.20**, **a.21**, **a.22**, **a.23**, **a.24** y **a.25**).

Sobre el **eje X**:

$$x' = x \quad \text{...(Ec. a.17)}$$
$$y' = y * \cos a_x - z * \sin a_x \quad \text{...(Ec. a.18)}$$
$$z' = y * \sin a_x + z * \cos a_x \quad \text{...(Ec. a.19)}$$

Donde: **x**, **y**, **z** = Punto de entrada.
a_x = Angulo de rotación.

Sobre el **eje Y**:

$$x' = x * \cos a_y + z * \sin a_y \quad \text{...(Ec. a.20)}$$
$$y' = y \quad \text{...(Ec. a.21)}$$
$$z' = -x * \sin a_y + z * \cos a_y \quad \text{...(Ec. a.22)}$$

Donde: **x**, **y**, **z** = Punto de entrada.
a_y = Angulo de rotación.

Sobre el **eje Z**:

$x' = x * cos\ a_z - y * sin\ a_z$ ...(Ec. a.23)
$y' = x * sin\ a_z + y * cos\ a_z$ ...(Ec. a.24)
$z' = z$ ...(Ec. a.25)

Donde: **x**, **y**, **z** = Punto de entrada.
a_z = Angulo de rotación.

a.2.4 Sesgo

En el espacio tridimensional, existen tres tipos de **sesgo**, uno por cada plano (**Ecs. a.26**, **a.27**, **a.28**, **a.29**, **a.30**, **a.31**, **a.32**, **a.33** y **a.34**).

Sobre el **plano XY**:

$x' = x$ ...(Ec. a.26)
$y' = y + a_y * z$ ...(Ec. a.27)
$z' = a_x * x + z$ ...(Ec. a.28)

Donde: **x**, **y**, **z** = Punto de entrada.
a_x = Constante de proporcionalidad en **x**.
a_y = Constante de proporcionalidad en **y**.

Sobre el **plano XZ**:

$x' = x + a_x * z$ ...(Ec. a.29)
$y' = y$ ...(Ec. a.30)
$z' = a_z * x + z$ ...(Ec. a.31)

Donde: **x**, **y**, **z** = Punto de entrada.
a_x = Constante de proporcionalidad en **x**.
a_z = Constante de proporcionalidad en **z**.

Sobre el **plano YZ**:

$x' = x + a_z * z$ ...(Ec. a.32)
$y' = y + a_y * z$ ...(Ec. a.33)
$z' = z$ ...(Ec. a.34)

Donde: **x**, **y**, **z** = Punto de entrada.
a_y = Constante de proporcionalidad en **y**.
a_z = Constante de proporcionalidad en **z**.

a.2.5 Proyecciones paralelas

Las **proyecciones paralelas**, permiten modelar la profundidad de los objetos, para que su apariencia sea más realista. Sobre el **eje Z = 0**. La proyección se realiza a un punto sobre el **eje Z**, por lo que el objeto sufre una deformación que tiende a juntar sus líneas en el punto **z = ¥** (**Ecs. a.35**, **a.36**, **a.37**, **a.34**, **a.39**, **a.40**, **a.41**, **a.42** y **a.43**).

$x' = x * (d + x) / z$ Para $z \neq 0$. ...(Ec. a.35)
$y' = y * (d + y) / z$ Para $z \neq 0$. ...(Ec. a.36)
$z' = z$ Para $z \neq 0$. ...(Ec. a.37)

$x' = x$ Para $z = 0$. ...(Ec. a.38)
$y' = y$ Para $z = 0$. ...(Ec. a.39)
$z' = z$ Para $z = 0$. ...(Ec. a.40)

Donde: **x**, **y**, **z** = Punto de entrada.
d = Distancia del origen al **Punto de Vista**.

Sobre el eje Z en infinito.

$x' = x * (d + |x|) / (z + d)$ ...(Ec. a.41)
$y' = y * (d + |y|) / (z + d)$ ...(Ec. a.42)
$z' = z$ ...(Ec. a.43)

Donde: **x**, **y**, **z** = Punto de entrada.
d = Distancia del origen al **Punto de vista**.

a.2.6 Proyecciones ortográficas

Las **proyecciones ortográficas** nos permiten visualizar las características del objeto visto desde cada uno de los ejes de coordenadas, es decir, es posible ver el frente, el lado y la parte de arriba de los objetos. Consiste en hacer cero las coordenadas del eje correspondiente y rotar los puntos de tal forma que la vista quede en el **plano XY**. Se tienen tres vistas ortográficas (**Ecs. a.44**, **a.45**, **a.46**, **a.47**, **a.48**, **a.49**, **a.50**, **a.51** y **a.52**).

Sobre el **eje x = 0**:

$x' = z * sin(90º)$ ...(Ec. a.44)
$y' = y$ ...(Ec. a.45)
$z' = 0$ ...(Ec. a.46)

Donde: **x**, **y**, **z** = Punto de entrada.

Sobre el **eje y = 0**:

$x' = x$ ...(Ec. a.47)
$y' = - z * sin(- 90º)$ ...(Ec. a.48)
$z' = 0$ ...(Ec. a.49)

Donde: **x**, **y**, **z** = Punto de entrada.

Sobre el **eje z = 0**:

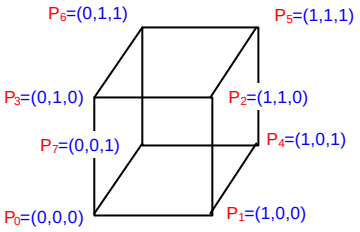
$x' = x$ ...(Ec. a.50)
$y' = y$ ...(Ec. a.51)
$z' = 0$ ...(Ec. a.52)

Donde: **x**, **y**, **z** = Punto de entrada.

a.2.7 Representación de objetos tridimensionales

Para poder realizar las operaciones de transformación en el espacio tridimensional, es necesario representar cada punto por sus coordenadas **(x,y,z)** y las rectas que los unen numerando los puntos y uniéndolos mediante sus índices como se muestra en la **Figura a.3**.

Puntos
(x,y,z)
0 0 0
1 0 0
1 1 0
0 1 0
1 0 1
1 1 1
0 1 1
0 0 1

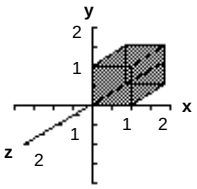


Líneas
(a ® b)
0 1
1 2
2 3
3 0
4 5
5 6
6 7
7 4
1 4
2 5
3 6
0 7

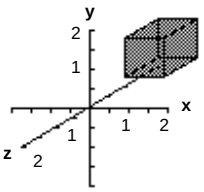
Figura a.3: Representación de **objetos tridimensionales**.

a.2.8 Operaciones con transformaciones tridimensionales

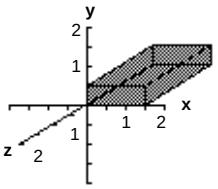
En la **Figura a.4** se observan los resultados de aplicar las transformaciones tridimensionales a un cubo.



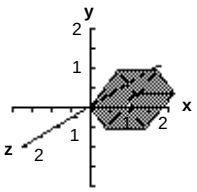
a) Objeto Original



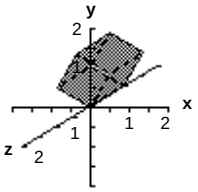
b) Traslación con $d_x = 2$, $d_y = 2$ y $d_z = 2$



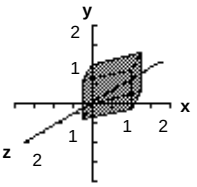
c) Escala con $s_x = 1.5$, $s_y = 0.5$ y $s_z = 2$



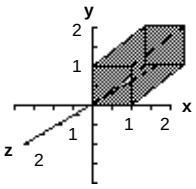
d) Rotación de $a_x = 60º$



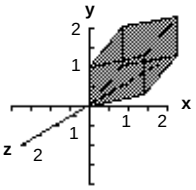
e) Rotación de $a_z = 60º$



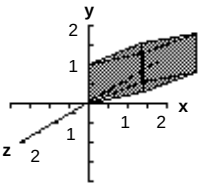
f) Rotación de $a_y = 60º$



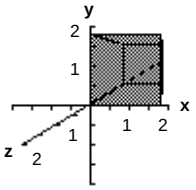
g) Sesgo en el **plano YZ (0.5,0.5)**



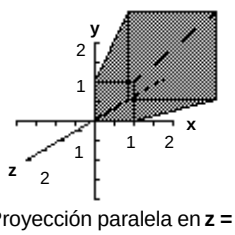
h) Sesgo en el **plano XY (0.5,0.5)**



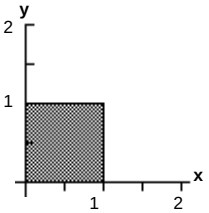
i) Sesgo en el **plano XZ (0.5,0.5)**



j) Proyección paralela en **z = ¥**



k) Proyección paralela en **z = 25**



l) Vista Ortogonal **z = 0**

Figura a.4: Resultados de las **transformaciones tridimensionales** en un **cubo**.

También es posible encontrar las vistas ortográficas de objetos complejos como se muestra en las **Figuras a.5**, **a.6**, **a.7** y **a.8**.

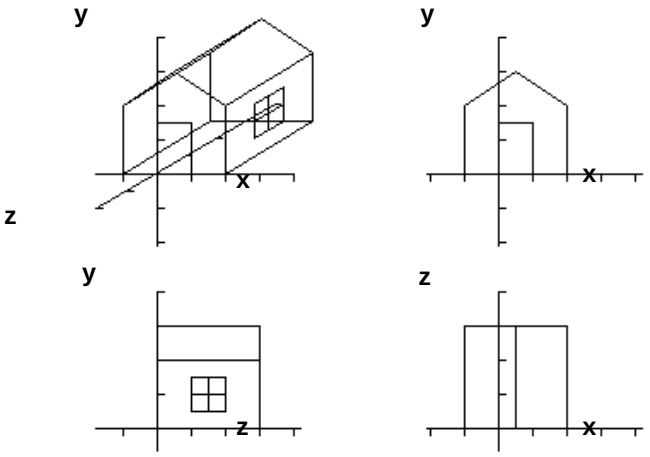


Figura a.5: Vistas de una **proyección paralela** de una **casa**.

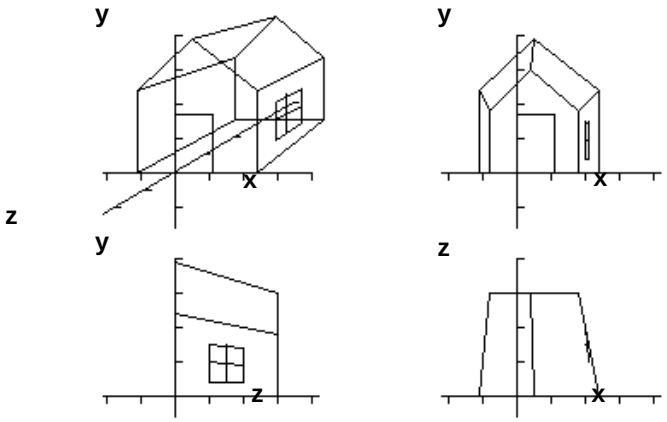


Figura a.6: Vistas de una **proyección en perspectiva** de una **casa**.

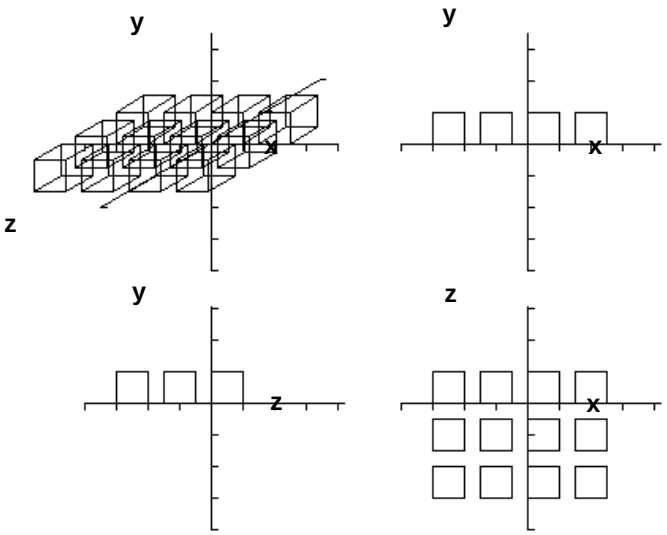


Figura a.7: Vistas de una **proyección paralela** de un conjunto de **cubos**

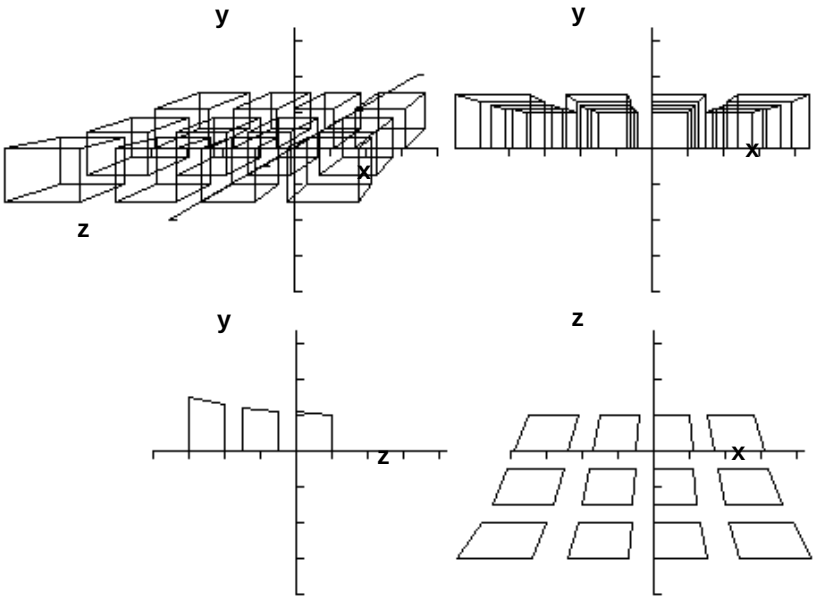


Figura a.8: Vistas de una **proyección en perspectiva** de un conjunto de **cubos**.

Apéndice B: Preguntas de Repaso

Lea las siguientes preguntas y subraye la respuesta que sea la más adecuada.

I. Introducción

1. El **objetivo** de **VRML 2.0** es la:

a) Descripción de entornos virtuales bidimensionales (**2D**).
b) Descripción de entornos virtuales tridimensionales (**3D**).
c) Descripción de entornos virtuales estáticos.
d) Ninguna de las anteriores.

2. Son **características** de **VRML 2.0**:

a) Semántica, poder descriptivo, interactividad, animación y modularidad.
b) Semántica, facilidad, contenido dinámico y especificación de geometrías.
c) Interacción con otras personas, semántica, animación y multiplataforma.
d) Ninguna de las anteriores.

3. De que tipo son los **archivos** de **VRML 2.0** y cuál es su **extensión**:

a) **ANSI** y `world`.
b) **ASCII** y `vrml`.
c) **ASCII** y `wrl`.
d) Ninguna de las anteriores.

4. ¿Cuál de las siguientes líneas es la **cabecera** para un archivo de **VRML 2.0**?

a) `#vrml v2.0 “utf8”`.
b) `#VRML V2.0 utf8`.
c) `#VRML “V2.0” UTF8`.
d) Todas las anteriores.

5. Un **nodo** es:

a) Un objeto geométrico tridimensional.
b) Una imagen.
c) Un color.
d) Todas las anteriores.

6. Los **campos** son:

a) Mensajes que circulan entre nodos.
b) Parámetros de los objetos.
c) Una lista de valores.
d) Todas las anteriores.

7. Las primitivas del campo `geometry` en el lenguaje de **VRML 2.0** son:

a) `Cubo`, `Esfera`, `Cono` y `Cilindro`.
b) `“box”`, `“sphere”`, `“cone”` y `“cylinder”`.
c) `Box`, `Sphere`, `Cone` y `Cylinder`.
d) Ninguna de las anteriores.

8. ¿Cuál es el campo por medio del cuál se define una **apariencia** o **textura**?

a) `appearance`.
b) `text`.
c) `material`.
d) Ninguna de las anteriores.

9. Para colocar la palabra `Texto 1` en **VRML 2.0** se emplea la siguiente línea:

a) `geometry Text { string ["Texto 1"] }`.
b) `geometry Texto { string ["Texto 1"] }`.
c) `geometry "Texto" { string [ Texto 1 ] }`.
d) Ninguna de las anteriores.

10. Para definir colores de cielo y tierra degradados se emplean:

a) `"frontURL"`, `"backURL"`, `"topURL"` y `"bottomURL"`.
b) `"skyAngle"`, `"sky"`, `"groundAngle"` y `"ground"`.
c) `skyAngle`, `skyColor`, `groundAngle` y `groundColor`.
d) Ninguna de las anteriores.

II. Transformaciones

1. Son **transformaciones básicas** en modelación tridimensional

a) Traslación, Inversión y Coloreado.
b) Translación, Vuelco y Acotación.
c) Translación, Rotación y Escala.
d) Ninguna de las anteriores.

2. Una translación en **VRML 2.0** se realiza a través del nodo:

a) `translation`.
b) `Transformación`.
c) `Shape`.
d) Todos los anteriores.

3. En la **rotación** de un objeto, el ángulo debe de especificarse en las siguientes unidades:

a) Grados.
b) Radianes.
c) Gradientes.
d) Todas de las anteriores.

4. El **escalamiento no uniforme** consiste en:

a) Cambiar el tamaño del objeto en una sola dirección ó en proporciones diferentes.
b) Cambiar el tamaño del objeto en todas las direcciones en la misma proporción.
c) Cambiar el tamaño del objeto en una cantidad impar.
d) Ninguna de las anteriores.

5. Cuando se **encadenan transformaciones**, el orden en que se aplican es:

a) De afuera hacia adentro.
b) Como aparecen.
c) De adentro hacia afuera.
d) Todas las anteriores.

6. Para que se pueden utilizar los **puntos de vista** en **VRML 2.0**:

a) Para moverse a partes importantes del mundo virtual.
b) Para enlazarse a otros mundos virtuales.
c) Para cambiar parámetros de un objeto.
d) Todas las anteriores.

7. Qué nodo permite definir **puntos de vista**:

a) `Vistas`.
b) `Viewpoint`.
c) `Descripción`.
d) Ninguna de las anteriores.

8. En un **punto de vista** el campo `description` define:

a) Un nombre que aparece como título en el navegador.
b) Una palabra.
c) Una etiqueta para identificar el punto de vista.
d) Ninguno de los anteriores.

9. En un **punto de vista**, la orientación esta definida por un vector **X Y** y **Z** y un **ángulo**, ¿Qué define el **vector**?

a) Posición donde se sitúa el punto de vista.
b) Posición de la etiqueta para identificar el punto de vista
c) Eje de rotación del punto de vista.
d) Ninguno de los anteriores.

10. El nodo que define un **ángulo de visión** es:

a) `fieldOfView`.
b) `Viewpoint`.
c) `View`.
d) Todos los anteriores.

III. Puntos, Líneas, Objetos y Mallas

1. Para que se emplea el nodo `PointSet`:

a) Para definir una línea.
b) Para definir puntos aislados.
c) Para definir puntos de vista.
d) Ninguno de los anteriores.

2. Para que se emplea el nodo `IndexedLineSet`:

a) Para definir objetos mediante líneas.
b) Para definir los puntos de una línea.
c) Para definir índices.
d) Ninguno de los anteriores.

3. Para que se emplea el nodo `IndexedFaceSet`:

a) Para definir los puntos de una línea.
b) Para definir puntos aislados.
c) Para definir vértices y caras de un objeto.
d) Ninguno de los anteriores.

4. En que dirección se refleja la luz **`IndexedFaceSet`**:

a) En la dirección paralela a la cara.
b) En un vector normal hacia afuera de la cara.
c) En un vector normal hacia adentro de la cara.
d) Ninguno de los anteriores.

5. El **color de cada cara** se determina a través del nodo:

a) **`colorPerVertex`**.
b) **`colores`**.
c) **`material`**.
d) Ninguno de los anteriores.

6. Para que se emplean las **mallas**:

a) Para definir figuras complejas.
b) Para describir valles.
c) Para hacer montañas.
d) Todas las anteriores.

7. Una **malla** se define de la forma:

a) Ángulos, Espacios y matriz de altura.
b) Distancias entre filas y columnas.
c) Dimensiones, Espacios y matriz de alturas.
d) Todas las anteriores.

8. El color en una **malla** se establece mediante el nodo:

a) **`geometry`**.
b) **`material`**.
c) **`Shape`**.
d) Ninguno de los anteriores.

9. Una figura generada por **extrusión** se define por:

a) Una sección de cruce.
b) Una línea en tres dimensiones (espina).
c) Una orientación.
d) Todas las anteriores.

10. La **extrusión** se emplea para:

a) Modelar las figuras básicas.
b) Modelar objetos mediante líneas.
c) Modelar con facilidad tubos, barras, etc.
d) Ninguno de los anteriores.

IV. Iluminación y Texturas

1. Son tres **formas de iluminación** empleadas en computadora:

a) Luz del sol, luz incandescente y luz fluorescente.
b) Luz direccional, luz puntual y luz Spot.
c) Luz de día, luz de noche y luz etérea.
d) Ninguno de los anteriores.

2. El nodo **`ambientIntensity`** determina:

a) En que proporción con la que el material refleja la intensidad ambiente.
b) En que proporción con la que el objeto emite luz.
c) La dirección de la luz ambiental.
d) Ninguno de los anteriores.

3. Para apreciar los **efectos de iluminación** en **VRML 2.0** se tiene que apagar:

a) La luz de fondo.
b) La luz que el objeto emite.
c) La luz de casco de minero.
d) Ninguno de los anteriores.

4. La **luz direccional** presenta la siguiente propiedad:

a) Irradia rayos en todas direcciones.
b) Todos sus rayos son paralelos.
c) Tiene dirección y ángulo.
d) Todas las anteriores.

5. La **luz puntual** presenta la siguiente propiedad:

a) Irradia rayos en todas direcciones.
b) Todos sus rayos son paralelos.
c) Tiene dirección y ángulo.
d) Todas las anteriores.

6. La **luz spot** presenta la siguiente propiedad:

a) Irradia rayos en todas direcciones.
b) Todos sus rayos son paralelos.
c) Tiene dirección y ángulo.
d) Todas las anteriores.

7. Las propiedades de **brillo**, **emisión** de luz y **transparencia** se definen en el nodo:

a) **`children`**.
b) **`translation`**.
c) **`material`**.
d) Ninguno de los anteriores.

8. Los **tipos de imágenes** que es posible usar como **textura** son:

a) **`BMP`**, **`DIB`** y **`JPG`**.
b) **`JPG`**, **`PNG`** y **`GIF`**.
c) **`MPEG`**, **`QVT`** y **`DIB`**.
d) Ninguno de los anteriores.

9. El nodo con el cuál se define una **imagen** como **textura** es:

a) **`ImageTexture`**.
b) **`url`**.
c) **`appearance`**.
d) Ninguno de los anteriores.

10. Cuando se aplica una **transformación**, que pasa con la **textura**:

a) Se aplica a la textura en si misma.
b) Se aplica al área del objeto que va a recibir la textura.
c) Se aplica a las coordenadas de la textura.
d) Ninguno de los anteriores.

V. Prototipos

1. Para poder definir un **nuevo nodo**, **VRML 2.0** cuenta con:

a) Comando **DEF**.
b) Nodo **PROTO**.
c) Comando USE.
d) Ninguno de los anteriores.

2. ¿Es posible definir **parámetros** en un **prototipo**?

a) Si.
b) No.
c) A veces.
d) Ninguno de los anteriores.

3. Para que sirve el nodo **Group**:

a) Para definir un grupo de nodos.
b) Para definir grupos de prototipos.
c) Para agrupar en un sólo objeto, varios subobjetos.
d) Todas las anteriores.

4. ¿Cuántos **objetos** es posible poner en un **prototipo**?

a) Uno.
b) Cualquier cantidad.
c) Una cantidad predefinida.
d) Ninguno de los anteriores.

5. La palabra que nos permite enlazar un **parámetro** es:

a) **LINK**.
b) **OF**.
c) **IS**.
d) Ninguno de los anteriores.

6. Cuando se define un **prototipo**, ¿Qué es necesario hacer para que exista?

a) Dejarlo como **PROTO**.
b) Instanciarlo a partir del prototipo.
c) Definir el prototipo.
d) Ninguno de los anteriores.

7. ¿Cómo se crea una **instancia a un prototipo**?

a) Colocando el nuevo nombre.
b) Definiendo un grupo.
c) Usando una primitiva básica.
d) Ninguno de los anteriores.

8. A un **prototipo** se le puede aplicar:

a) Transformaciones.
b) Nuevos parámetros (si los define).
c) Nada.
d) Todas las anteriores.

9. El nodo **`DEF`** permite:

a) Definir un prototipo.
b) Dar nombre a cierto nodo.
c) Utilizar una definición.
d) Ninguno de los anteriores.

10. El nodo **`USE`** permite:

a) Definir un prototipo.
b) Dar nombre a cierto nodo.
c) Utilizar una definición.
d) Ninguno de los anteriores.

VI. Anchors, Billboarding y Colisiones

1. Cuál es el nodo que permite en **VRML** realizar **enlaces** (links) a otros documentos:

a) **`Anchor`**.
b) **`Enlace`**.
c) **`Link`**.
d) Ninguno de los anteriores.

2. De que forma **VRML** indica que un objeto esta **activo** como enlace:

a) Intensifica el objeto.
b) Cambia de forma el cursor.
c) Emite un sonido.
d) Ninguno de los anteriores.

3. Es posible emplear los **puntos de vista** (**`Viewpoint`**) como **`Anchors`**:

a) No.
b) No sé.
c) Si.
d) Ninguno de los anteriores.

4. Cómo se **inserta** un mundo virtual existente en otro mundo virtual:

a) No es posible.
b) Con **`Inline`**.
c) Con **`Anchor`**.
d) Ninguno de los anteriores.

5. Para que se emplea el **`Billboard`**:

a) Para representar objetos complejos.
b) Para representar objetos simples.
c) En lugar de usar polígonos.
d) Ninguno de los anteriores.

6. **VRML** permite detectar **colisiones** entre:

a) Objetos.
b) Observador y objetos.
c) No lo permite.
d) Ninguno de los anteriores.

7. ¿Cómo se activa la **detección de colisiones** en **VRML**?

a) `Colisión TRUE`.
b) `collide TRUE`.
c) `collide FALSA`.
d) Ninguno de los anteriores.

8. ¿Qué es un **evento** en **VRML**?

a) Una acción del ratón.
b) Un movimiento.
c) Un mensaje que puede ser enviado por un objeto y recibido por otro.
d) Ninguno de los anteriores.

9. ¿Cómo deben de ser los **eventos** para que puedan funcionar?

a) Del mismo tipo.
b) De tipos distintos.
c) No importa el tipo.
d) Ninguno de los anteriores.

10. Mediante que **nodo** se **enlazan dos eventos** (uno de entrada y uno de salida):

a) Mecanismo `ROUTE`.
b) Mediante una ruta.
c) `Link`.
d) Ninguno de los anteriores.

VII. Sensores e Interpoladores

1. ¿Cuándo se pone en marcha el **sensor de tiempo** (`TimeSensor`)?

a) Al iniciarse el navegador.
b) Al iniciar el sistema.
c) Al iniciar un enlace.
d) Ninguno de los anteriores.

2. ¿Qué tipo de **interpolación** emplea **VRML**?:

a) De Newton.
b) Binomial.
c) Lineal.
d) Ninguno de los anteriores.

3. ¿Cuántos tipos de **interpoladores** tiene **VRML**?:

a) 2.
b) 4.
c) 6.
d) Ninguno de los anteriores.

4. ¿Cuál es la función de **key** en un **interpolador**?:

a) Definir una llave de acceso.
b) Dar el porcentaje de cambio.
c) Proporcionar una ayuda al usuario.
d) Ninguno de los anteriores.

5. Para que se emplea el **sensor de proximidad**:

a) Seguimiento de los movimientos del usuario.
b) Para aproximarse a un objeto.
c) Como un punto de vista.
d) Ninguno de los anteriores.

6. En donde se **centra** un **sensor de proximidad**:

a) En la esquina superior izquierda del sistema de coordenadas.
b) En la parte posterior del sistema de coordenadas.
c) En el centro del sistema de coordenadas.
d) Ninguno de los anteriores.

7. Para que se emplea el **sensor de tacto**:

a) Para detectar cuándo el observador apunta a un objeto.
b) Para enviar la posición de un observador.
c) Para imponer límites.
d) Ninguno de los anteriores.

8. El **sensor de visibilidad** sirve para:

a) Para simular desvanecimiento.
b) Para tapar la vista de un observador.
c) Optimización.
d) Ninguno de los anteriores.

9. Cuantos tipos de **sensores de movimiento** hay:

a) 1.
b) 2.
c) 3.
d) Ninguno de los anteriores.

10. Cuáles son lo tipos de **sensores de movimiento**:

a) Punto, Línea y Plano.
b) Plano, Cilindro y Esfera.
c) Cubo, Cono y Esfera.
d) Ninguno de los anteriores.

VIII.Sonido

1. ¿Qué tipos de **sonido** maneja **VRML**?:

a) Ambiental y Espacial.
b) Mono y Estereofónico.
c) Sonido **5.1**
d) Ninguno de los anteriores.

2. ¿Qué es el **sonido espacial**?:

a) El que se encuentra en el espacio.
b) El que varía según la posición del espectador.
c) El que se escucha igual en todo el espacio.
d) Ninguno de los anteriores.

3. ¿Qué es el **sonido ambiental**?:

a) El que se encuentra en el espacio.
b) El que varía según la posición del espectador.
c) El que se escucha igual en todo el ambiente virtual.
d) Ninguno de los anteriores.

4. ¿Cómo se define el **sonido** en **VRML**?:

a) Nodo **`sonido`**.
b) Nodo **`Audio`**.
c) Nodo **`Sound`**.
d) Ninguno de los anteriores.

5. ¿Cómo se activa el **sonido espacial** en **VRML**?:

a) **`ambiente FALSE`**.
b) **`spatialize TRUE`**.
c) **`espacial TRUE`**.
d) Ninguno de los anteriores.

IX. Nivel de Detalle

1. ¿Qué es el **nivel de detalle** (**LOD**)?:

a) Es una técnica empleada para mantener el refresco de la pantalla en tiempos óptimos.
b) Permite predecir lo que verá el observador.
c) Oculta detalles del mundo al observador.
d) Todos los anteriores.

2. ¿Cómo opera el **nivel de detalle** en **VRML**?:

a) Definiendo espacios o rangos delante del observador.
b) Colocando obstáculos delante del observador.
c) Detectando colisiones del observador.
d) Ninguno de los anteriores.

3. ¿Para qué se emplea el campo **`level`** en el **nivel de detalle**?

a) Para definir objetos.
b) Para ver los objetos.
c) Para colocar las diferentes versiones de los objetos.
d) Ninguno de los anteriores.

4. Para que se emplea la información de navegación:

a) Para colocar datos del o los autores del mundo.
b) Para poner el titulo del mundo.
c) Para colocar las características físicas del avatar y el tipo de controles del navegador.
d) Todos los anteriores.

5. En la información del navegador puede definirse la **entidad virtual** (**avatar**):

a) Si.
b) No.
c) No sé.
d) Ninguno de los anteriores.

X. JavaScript

1. ¿Qué partes constituyen un nodo `Script`?

a) Definición de campos y eventos, y el código del lenguaje.
b) Campos y eventos.
c) Eventos de entrada y salida.
d) Ninguno de los anteriores.

2. Toda función de **JavaScript** tiene asociados dos elementos:

a) Un valor de entrada y uno de salida.
b) Un nombre de entrada y una función de salida.
c) Valor del elemento de entrada y tiempo.
d) Ninguno de los anteriores.

3. ¿Cómo se usa un `eventOut` en **VRML**?:

a) Asignándole un valor desde adentro de la función.
b) Enviándolo como objeto a otra función.
c) No hace nada.
d) Ninguno de los anteriores.

4. Para que sirven los campos en un `script`:

a) Para recibir eventos.
b) Como variables globales.
c) Para comparar objetos.
d) Ninguno de los anteriores.

5. Para que se emplea el campo `mustEvaluate`:

a) Para realizar una evaluación de la función.
b) Para comparar campos.
c) Para evaluar el `script` cada vez que se genere un evento.
d) Ninguno de los anteriores.

XI. VRML y Java

1. ¿Por qué emplear **Java** con **VRML**?:

a) Para complementar a **VRML** en cálculos complejos.
b) Para graficar mejor.
c) Para interactuar con los objetos.
d) Ninguno de los anteriores.

2. Como se logra la interacción con **Java** y **VRML**

a) Compilando **VRML** en **Java**.
b) Mediante la **External Authoring Interface** (**EIA**).
c) Insertando ambos en una página **HTML**.
d) Ninguno de los anteriores.

3. ¿Qué permite la **External Authoring Interface** (**EIA**)?:

a) La comunicación entre ambos lenguajes.
b) El paso de parámetros entre mundos virtuales y **Java**.
c) Integrar **Java** y **VRML**.
d) Todos de los anteriores.

4. ¿Cómo se agrega un nodo desde **Java** a **VRML**?:

a) Sin hacer nada.
b) Con **`setvalue`**.
c) Con **`Link`**.
d) Todos de los anteriores.

5. Para ejecutar un programa con **Java** y **VRML** es necesario:

a) Sólo el navegador.
b) Sólo un visualizador de **VRML**.
c) Interprete de **HTML**, la máquina virtual de **Java** y el código **VRML**.
d) Todos de los anteriores.

XII. Introducción a X3D

1. ¿Qué es **X3D**?:

a) Un lenguaje de programación de uso general.
b) Un lenguaje de programación para gráficos vectoriales.
c) Un lenguaje de procesamiento de texto.

2. De que tipo son los **archivos** de **X3D** y cuál es su **extensión**:

a) **ANSI** y **`Extended`**.
b) **ASCII** y **`x3d`**.
c) Ninguna de las anteriores.

3. Cuál de las siguientes líneas es la **cabecera** para un archivo de **X3D**:

a) **`<?xml versión="1.0" encoding="UTF-8">`**.
b) **`#X3D "utf8"`**.
c) Todas las anteriores.

4. ¿Cuál es el campo por medio del cuál se define una **apariencia** o **textura**?

a) **`Appearance`**.
b) **`text`**.
c) Ninguna de las anteriores.

5. Para colocar la palabra **`Texto 1`** en **X3D** se emplea la siguiente línea:

a) **`geometry Text { string ["Texto 1"] }`**.
b) **`<Text string='"Texto 1"'>`**.
c) Ninguna de las anteriores.

Referencias

[Alarcón 00] De Alarcón Álvarez, E. y Parés I. Burgués, N.; **"Manual Práctico de VRML 2.0"**; Ed. Biblioteca Técnica de Programación, España, **2000**, **187pp**.

[Alchemy 96] **"Image Alchemy 1.8"**; Handmade Software, U.S.A., **1996**, **226pp**.

[Deitel 98] Deitel, H.M. y Deitel, P.J.; **"Como programar en Java"**; Ed. Prentice-Hall, México, **1998**, **1056pp**.

[Foley 96] Foley, James D., van Dam, Andreis, Feiner, Steven K., Hughes, Jonh F. y Phillips, Richard L.; **"Introducción a la Graficación por Computador"**; Ed. Addison Wesley Iberoamericana, U.S.A., **1996**, **650pp**.

[Johnson 87] Johson, Nelson; **"Advanced Graphics in C: Programming and Techniques"**; Ed. McGraw-Hill, U.S.A., 1987, **669pp**.

[Shneiderman 98] Shneiderman, Ben; "**Designing the Users Interface, Strategies for effective Human-Computer Interaction**"; 3ra edición, Ed. Addison-Wesley, U.S.A., **1998**, **638pp**.

[VRML 97] **"The Virtual Reality Modeling Language. International Standard ISO/IEC 14772-1 1997"**; Internet: `www.vrml.org/Specifications/ VRML`, **1997**, **236pp**.

[X3D 04] **"Extensible 3D (X3D): ISO/IEC 19775:2004"**; Internet: `www.web3d.org`.

NOTAS