FONDAMENTI DI VRML 2.0 'MOVING WORLDS'

Alessandro Demontis

1. Introduzione

VRML è l' acronimo di *Virtual Reality Modeling Language*, ossia Linguaggio di Modellazione della Realtà Virtuale. VRML non è un linguaggio di programmazione vero e proprio, bensì un linguaggio di Mark-up, un metalinguaggio, proprio come lo sono HTML ed XML.

VRML nasce da una idea del programmatore Mark Pesce, all' inizio degli anni '90, per soddisfare una esigenza di interattività e di trasposizione concettuale da 2D a 3D.

Questo linguaggio conobbe un boom intorno al 1995/98, soprattutto grazie alla sua flessibilità, alla nascita di VRMLscript, un linguaggio simile a javascript e quindi meno legato ai tags, con veri e propri comandi di condizioni, di manipolazione, di creazione di figure, e di controllo di condizioni ed eventi.

Ciò che differenzia un 'mondo virtuale' creato con VRML da uno creato, ad esempio, con 3DstudioMax, è la possibilità dell' utente di 'navigare' nell' ambiente creato ed interagire con esso tramite degli eventi programmati.

L' avvento della tecnologia OLE, dei controlli ActiveX e la loro capacità di funzionare in computer meno prestanti e dotati di minore banda rispetto a quelli richiesti per VRML, resero questo linguaggio sconveniente per la modellazione 3D.

Tuttora il VRML (o Vermal, come viene chiamato spesso nell' ambito della programmazione) viene utilizzato quasi esclusivamente in ambiente scientifico, medico, meteorologico, e in coppia con Java in applicazioni distribuite inerenti simulazioni.

2. VRML e il Web

L' applicazione primaria di VRML fu quella di portare nel tridimensionale il mondo del web. Pensiamo per esempio a un sito che dalla pagina principale conduca ad una pagina-catalogo di prodotti, as una pagina per contattare l' azienda produtrice, e sa una pagina di links ad altri siti di aziende del settore.

Con VRML potremmo creare una casa, con sulla porta una insegna recante il logo della azienda relativa al sito.

Entrando nella casa, avremmo 3 stanze: nella prima potremmo avere una serie di 'quadri' dei vari prodotti. Clikkando su ogni quadro potrà aprirsi un video dimostrativo del progetto. Potremmo addirittura, anzi che usare i quadri, creare modelli in 3D dei vari prodotti e disportli

in una mensola. Il visitatore potrebbe 'toccare' ogni prodotto, girarlo, clikkarci per esempio per far partire un filmato dimostrativo.

Ora spostiamoci nella seconda stanza: potremmo trovare al centro un tavolino in 3D con sopra un foglio e una penna. Clikkando sul foglio, ci apparirebbe davanti agli occhi un form da compilare, e dopo, riclikkandoci, un controllo CGI spedirebbe il nostro form all' email della azienda. Nella terza pagina potremmo trovare, appesi al muro come quadri, i vari banners delle aziende associate a quella che stiamo visitando.

Fantastico vero?!

3. Progettazione di Mondi virtuali

Un mondo virtuale può essere creato in due modi:

- utilizzando un file ASCII in cui verranno scritti a mano i vari comandi, come per esempio si usa fare nella programmazione batch;
- utilizzando editors visuali o semi visuali (ne esistono diversi con vari livelli di interazione pre-programmata).

La soluzione migliore, generalmente, è costruire le parti grafiche con un editor visuale, e poi utilizzare un editor semi visuale in cui possa essere inserito codice a mano.

I mondi virtuali per poter essere visualizzati necessitano di un plug-in da installare ed associare a un browser web, o di un browser VRML dedicato.

Uno dei primi esempi di Editor VRML fu Paragraph Homespace Builder, successivamente acquistato da Cosmo Software, che produsse Homespace Design ed il plugin di visualizzazione Cosmo Player.

Vediamo ora un esempio di file VRML:

```
#VRML V2.0 utf8
# Sfera rossa
Shape {
  appearance Appearance {
    material Material { emissiveColor 1 0 0 }
  }
  geometry Sphere { radius 3 }
}
```

Questo listato produce in un browser web con plug-in VRML una sfera rossa.

La dichiarazione: **#VRML V2.0 utf8** comunica al browser che deve indirizzare il file al plugin anzichè interpretarlo direttamente. La parte successiva contenente i comandi è divisa in **Nodi** e **Fields**:

L' istruzione **appearance**, con prima lettera minuscola, è un **Field** , mentre **Appearance** con prima lettera minuscola è un **Nodo**.

Un Nodo è in generale una entità di programmazione che definisce un 'oggetto' sia esso fisico o meno.

Per capire la differenza, consideriamo il primo nodo utilizzato: **Shape**.

Definiamo il nodo Shape come una entità contenente due 'caratteristiche' o fields:

- **appearance**, che stabilisce le qualità e proprietà fisiche dell' oggetto che dobbiamo creare;
- **geometry**, che stabilisce la forma dello stesso tramite **primitive** geometriche, in questo caso **Sphere**.

Ognuno di questi fields, a sua volta, contiene un nodo che ha lo stesso nome.

Così il field *appearance* contiene il nodo *Appearance*, che a sua volta contiene un field **material**, che contiene un nodo **Material**.

L' uso delle parentisi stabilisce la relazione tra nodi e fields.

In sostanza, prima stabiliamo che l' *'appearance'* del nostro oggetto Shape avrà un materiale con una proprietà di colore rosso (emissivecolor 1 0 0 - dove 1 è il codice per il rosso, 0 per il verde e 0 per il blu), poi stabiliamo che la '*geometry*' dell' oggetto sarà una sfera di raggio 3 (Sphere {radius 3}).

Salviamo il nostro file ASCII con estensione **WRL** e trasciniamolo nel browser (ovviamente dopo aver installato il plug-in).

Il risultato sarà questo:

Dimentichiamoci per ora del fatto che più che una sfera sembri un poligono a 12 lati, l' importante ora è capire il meccanismo.

4. Ragionare in 3D

Per poter costruire con VRML qualcosa di più complicato, dobbiamo imparare a ragionare in tre dimensioni. Il 'mondo' virtuale è uno spazio in cui il punto 0 delle coordinate, chiamato Origine e avente coordinate (0 0 0), è rappresentato dal centro dello spazio visibile del mondo.

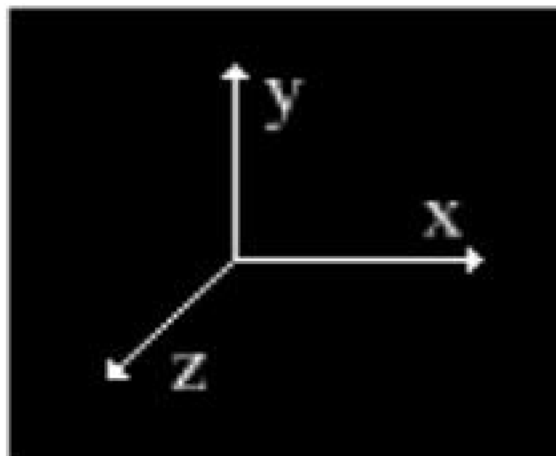
Se noi non stabiliamo un diverso punto origine, ogni oggetto verrà disegnato al centro del nostro mondo.

Questo fa anche sì che, se noi definiamo come nell' esempio precedente una sfera di raggio 3, il raggio partirà dal centro dello schermo.

Consideriamo ora, anziché una sfera, di dover disegnare un cilindro.

Un cilindro è un oggetto dotato di un raggio e di una altezza.

Data la disposizione degli assi:



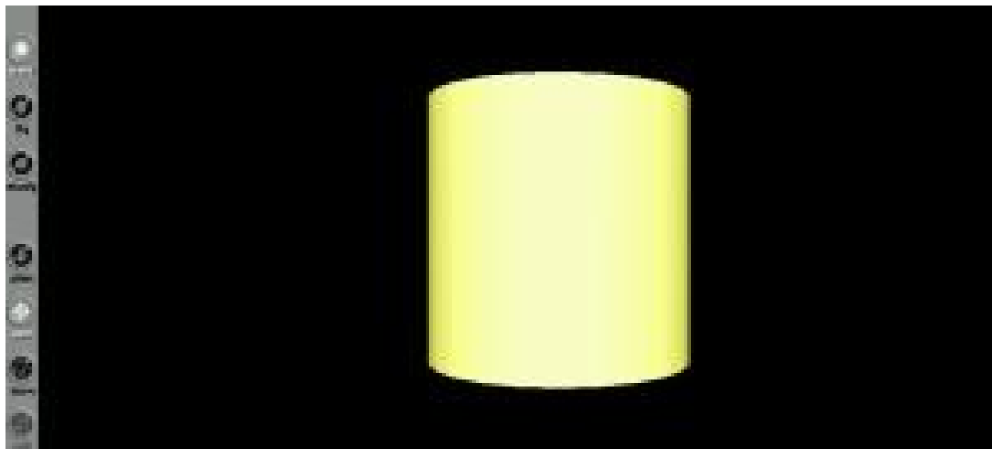
Un cilindro di altezza 8, si eleverà dall' origine di 4 unità lungo l' asse **Y positivo** e continuerà dall' origine di 4 unità verso l' asse **Y negativo**.

Scriviamo il nostro programmino.

Il nodo da utilizzare è sempre **Shape**, ma stavolta come geometry useremo **Cylinder** anziché Sphere:

```
#VRML V2.0 utf8
# Cilindro
Shape {
  appearance Appearance {
    material Material {   emissiveColor .5 .5 0  }
  }
  geometry Cylinder {
    radius 2
    height 4
  }
}
```

Salviamo ancora in formato wrl e carichiamo nel browser:



Il risultato appunto è un cilindro, stavolta di colore giallo, che và in altezza dal punto Y=-4 a Y=+4, in larghezza da X=-1 a X=+1 e in profondità da Z=-1 a Z=+1.

Cosa succede se noi uniamo i due nodi Shape nello stesso file wrl?

Al lato pratico si manifesteranno 2 problemi.

Il fatto che il nodo Shape venga utilizzato come 'contenitore' per le istruzioni sia della palla sia del cilindro, creerà un conflitto se non usiamo un particolare accorgimento che tra breve spiegherò.

Inoltre, per il discorso che abbiamo appena visto delle coordinate, gli oggetti si sovraporranno l' un l' altro, perché il centro di entrambi coinciderà con l' origine degli assi.

Il primo problema si risolve con il nodo **Transform** e il field **Children**, che permettono di 'annidare' tra loro due definizioni di oggetti, una subordinata all' altra.

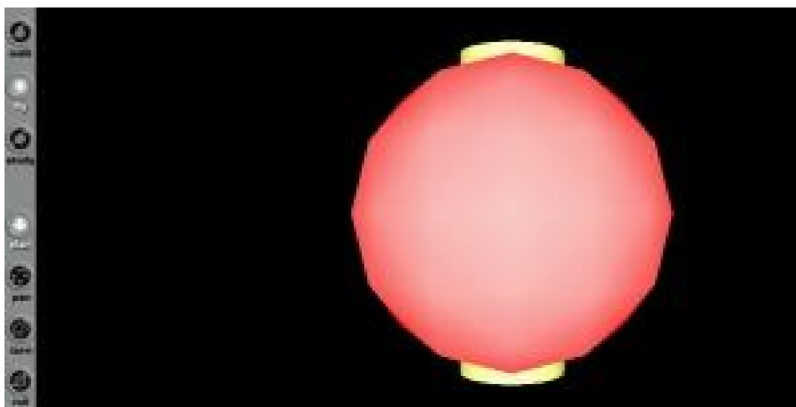
In sostanza il nodo Transform avverte il plug-in che si sta per trattare un secondo oggetto con una qualche relazione con il primo.

Modifichiamo il nostro codice o creiamo un nuovo file in cui ‘innestiamo’ i due oggetti:

```
#VRML V2.0 utf8
# Sfera rossa più cilindro giallo
Shape {
  appearance Appearance {
    material Material { emissiveColor 1 0 0 }
  }
  geometry Sphere { radius 3 }
}
Transform {
children [
Shape {
  appearance Appearance {
    material Material {   emissiveColor .5 .5 0  }
  }
  geometry Cylinder {
    radius 1
    height 6
  }
}
]
}
```

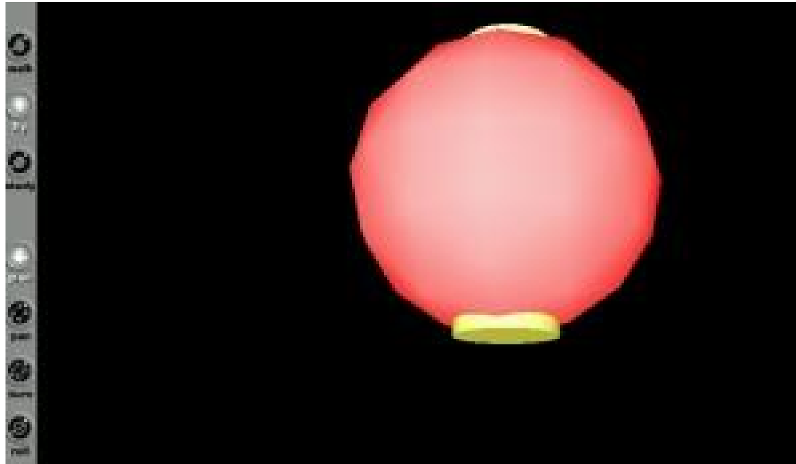
Qui ho modificato le dimensioni delle primitive Sphere e Cylinder, altrimenti la sfera avrebbe completamente ricoperto il cilindro.

Il risultato a video di questo listato è il seguente:



Come potete vedere, il cilindro ‘spunta’ in verticale sopra e sotto la sfera.

L’ evidenza di questa ‘intrusione’ dei due oggetti è più evidente se ruotiamo un po’ gli stessi:



Questo può tornare utile, certo, per esempio nella creazione di nuove forme, ma poteva non essere il risultato voluto.

E se avessimo. per esempio. voluto che la sfera fosse a contatto **sopra** il cilindro?

Ci viene in aiuto il field ‘**translation**’ del nodo Transform.

Ricapitolando, abbiamo un cilindro di raggio=1 e altezza=6, e una sfera di raggio=3. Partendo dal punto d’ origine con coordinate 0, il cilindro si eleverà dall’ origine di 3 unità. Sarà quindi 3 il punto sulle Y positive in cui la sfera dovrà poggiare .

Per fare ciò dovremo fare due cose:

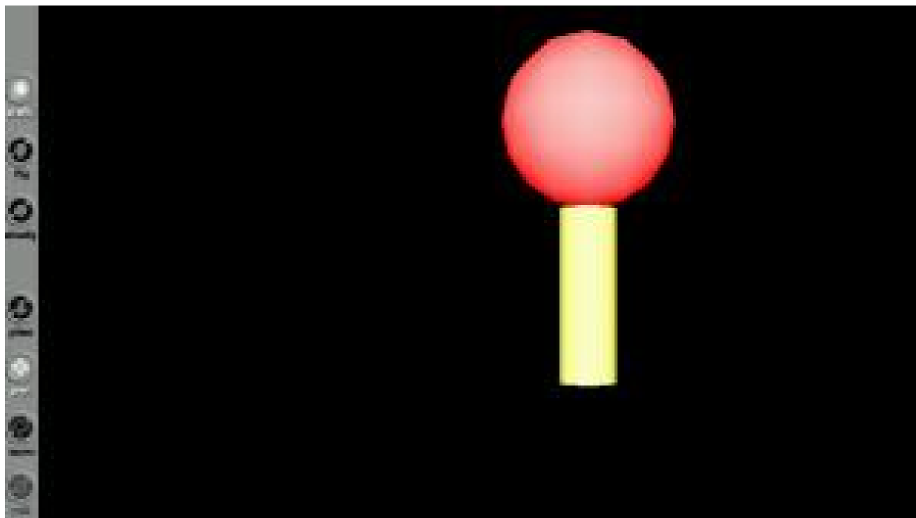
- rendere la sfera ‘dipendente’ dal cilindro e non, come è attualmente, il contrario;
- effettuare una traslazione di 6 unità verso l’ alto prima di creare la sfera. Perché 6? Perché la sfera ha raggio 3, quindi si estende dal suo centro, lo 0 relativo, di 3 unità verso il basso. Queste 3 unità dovranno essere aggiunte al 3 che rappresenta il punto più alto del cilindro.

Il listato risultante sarà il seguente:

```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material {  emissiveColor .5 .5 0  }
```

```
  }
  geometry  Cylinder { radius 1 height 6 }
 }
 Transform {
  translation 0 6 0
  children [
   Shape {
  appearance Appearance {
   material Material { emissiveColor 1 0 0 }
  }
  geometry Sphere { radius 3 }
 }
 ]
 }
```

Ed il risultato a schermo, allontanando un po' gli oggetti per riprenderli completamente, sarà il seguente:



Provate ora a creare un secondo file in cui la sfera e il cilindro siano affiancati, entrambi allineati con l' origine, e con la sfera a 2 unità a destra del cilindro.

Il field translation stavolta conterrà una traslazione lungo l' asse X.

Ma di quanto? Calcoliamo.

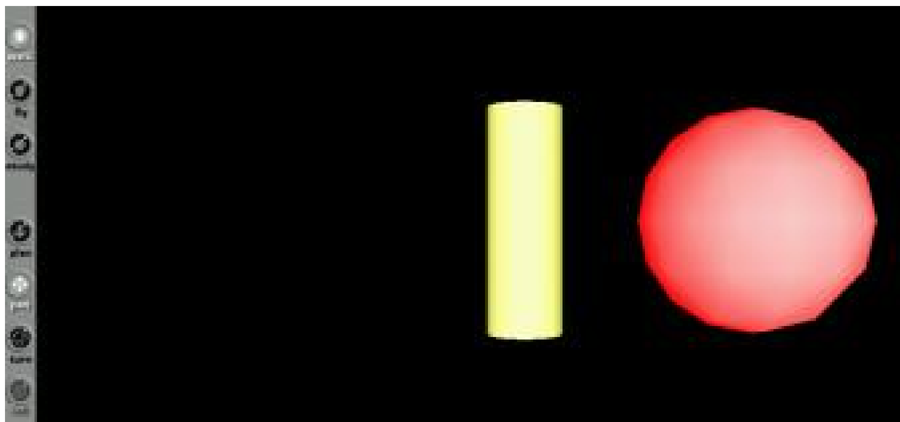
Il cilindro ha raggio=1, quindi dal suo centro (che corrisponde all' origine), il lato destro avrà termine alla coordinata X=1. Aggiungiamo le 2 unità di distanza, e anche le 3 unità di raggio della sfera. In tutto avremo 6 unità. Il field translation quindi avrà questi parametri:

translation 6 0 0

Il listato sarà quindi:

```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material {  emissiveColor .5 .5 0  }
  }
  geometry  Cylinder {
radius 1
height 6
}
}
Transform {
  translation 6 0 0
  children [
    Shape {
  appearance Appearance {
    material Material { emissiveColor 1 0 0 }
  }
  geometry Sphere { radius 3 }
}
]
}
```

E il risultato, riprendendo gli oggetti interamente allontanandoli, il seguente:



Possiamo qui notare una particolarità: mentre il cilindro ci appare esattamente frontale, la sfera, trovandosi spostata di lato, ci viene disegnata secondo le leggi della prospettiva centrale. Il suo bordo destro è allungato.

Se proviamo a spostare gli oggetti verso sinistra vedremo gradualmente la sfera riprendere la sua forma consueta, e il cilindro alterarsi.

5. Spostamento nello spazio e rotazione.

Finora abbiamo visto due oggetti disposti uno sopra l' altro, inglobati l' uno nell' altro, o affiancati. Il comune denominatore di questi esercizi è la posizione centrale che essi mantengono rispetto all' origine degli assi. Detta chiaramente, tutti gli oggetti sono sullo stesso piano, allineati con l' origine e disposti frontalmente.

Proviamo ora a costruire due oggetti che siano distanti tra loro non lungo gli assi X o Y, ma lungo l' asse Z, cioè nel senso della profondità.

In sostanza, creeremo due oggetti in cui uno è più 'arretrato' rispetto all' altro.

Utilizzeremo stavolta un cilindro e un parallelepipedo.

La prima parte del nostro listato sarà:

```
#VRML V2.0 utf8
# Cilindro con parallelepipedo arretrato
Shape {
  appearance Appearance {
    material Material {   emissiveColor .5 .5 0  }
  }
  geometry Cylinder {
    radius 2
    height 4
  }
}
```

Come abbiamo visto, questo creerà il cilindro giallo.

Prima di costruire il parallelepipedo dobbiamo definire la sua traslazione nell' asse Z, cioè dobbiamo portalo **dietro** il cilindro. Se non lo spostiamo anche un po' su un lato, il parallelepipedo apparirà centrato e quindi nascosto dal cilindro.

Assegnamo quindi a translation un movimento di -10 unità lungo l' asse Z e di 4 unità lungo l' asse X:

```
Transform {
  translation 4 0 –10
```

Ora possiamo costruire il nostro parallelepipedo.

Per questo usiamo un' altra primitiva: **Box**

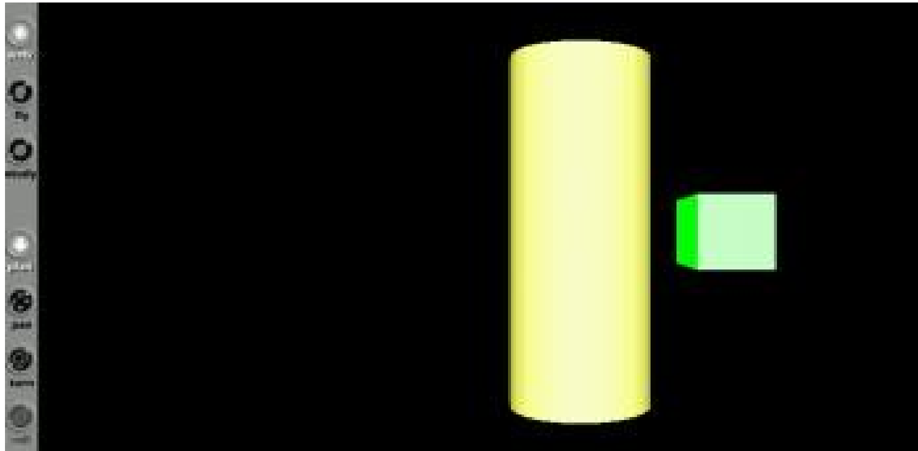
Box accetta come parametro il campo 'size' seguito dalle dimensioni in unità delle singole facce lungo i 3 assi.

```
children [
   Shape {
  appearance Appearance {
    material Material { emissiveColor 1 0 0 }
  }
  geometry Box { size 2 2 4 }
}
]
```

Ricordiamoci sempre che dopo un Transform bisogna stabilire la relazione Childern prima di definire la nuova Shape. Questa darà forma a un parallelepipedo alto 2 unità, largo 2 unità, e profondo 4 unità. Non dimentichiamoci poi di chiudere il Transform con la sua partentisi grafa, e il listato finale sarà il seguente:

```
#VRML V2.0 utf8
# cilindro giallo in primo piano con dietro un parallelepipedo verde
Shape {
  appearance Appearance {
    material Material {  emissiveColor .5 .5 0  }
  }
  geometry  Cylinder {
    radius 1
    height 5
  }
}
Transform {
  translation 4 0 -10
  children [ Shape {
  appearance Appearance {
    material Material { emissiveColor 0 1 0 }
  }
  geometry Box { size 2 2 4 }
}]
}
```

Il cui risultato a video sarà:



Noterete anche qui, come per la sfera affiancata al cilindro, che il parallelepipedo appare 'ruotato' a causa della prospettiva centrale.

Invertiamo ora le posizioni, e poniamo il parallelepipedo al centro e il cilindro arretrato e spostato lateralmente. Per rendere meglio la prospettiva diamo le stesse dimensioni al parallelepipedo e al cilindro e aggiungiamo 2 unità di distanziamento verso destra:

```
#VRML V2.0 utf8
# parallelepipedo giallo in primo piano con dietro un cilindro verde
Shape {
  appearance Appearance {
    material Material {  emissiveColor .5 .5 0  }
  }
  geometry Box { size 3 3 5 }
}
Transform {
  translation 6 0 -10
  children [
    Shape {
  appearance Appearance {
    material Material { emissiveColor 0 1 0 }
  }
  geometry Cylinder {
    radius 1
    height 3
  }
```

```
}
]
}
```

Ecco il risultato:

Stavolta il parallelepipedo, essendo al centro, apparirà come un semplice quadrato perché visto di fronte.

Parliano adesso di rotazioni.

Vogliamo fare in modo che il parallelepipedo, pur stando al centro, venga mostrato leggermente ruotato, così da poterne constatare la solidità e la prospettiva.

Ci viene in aiuto il field '**rotation**' del nodo Transform.

La sua sintassi è: **rotation 0 1 0 1.57**

Le prime 3 cifre rappresentano gli assi lungo cui effetuare la rotazione. Da ricordare che, quando usiamo rotation, gli assi vengono invertiti, quindi la prima cifra rappresenta l' asse Y, la seconda l' asse X, e la terza l' asse Z.

L' ultima cifra è l' angolo di rotazione espresso in **radianti**.

L' istruzione scritta come sopra dichiara che la rotazione avverrà lungo l' asse X positivo, cioè verso destra, e di una angolazione in radianti uguale a 1.57, che tramite la formula matematica: **radianti= gradi*PIgreco/180** ci restituisce 90 gradi.

Possiamo provare ad applicare una rotazione, per esempio di 45 gradi verso destra (0,785 radianti), al parallelepipedo.

Per farlo, prima di definire la Shape per il parallelepipedo, dovremo inserire un nuovo Transform contenente la rotazione ed inserire la Shape in un campo children:

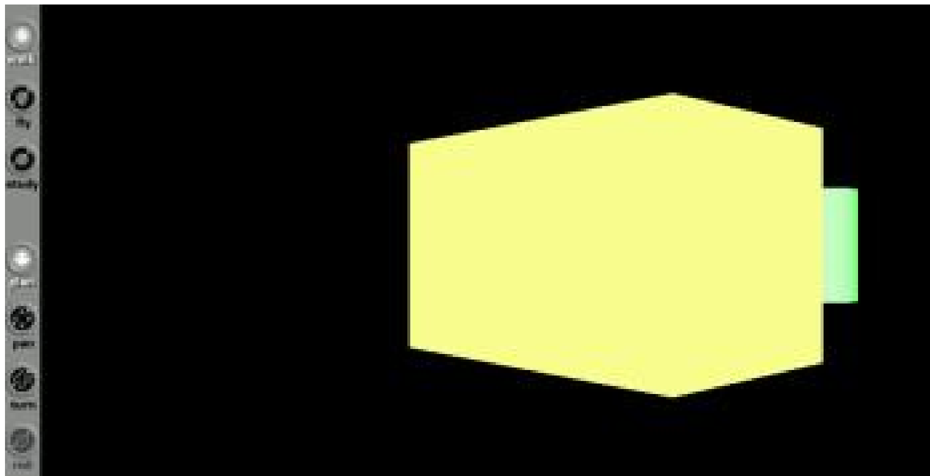
```
Transform {
  rotation 0 1 0 0.785
  children [
```

e il listato completo sarà:

```
#VRML V2.0 utf8
# parallelepipedo giallo ruotato con dietro un cilindro verde
Transform {
  rotation 0 1 0 0.785
  children [
Shape {
  appearance Appearance {
    material Material {  emissiveColor .5 .5 0  }
  }
  geometry Box { size 3 3 5 }
}
]
}
Transform {
  translation 6 0 -10
  children [
    Shape {
  appearance Appearance {
    material Material { emissiveColor 0 1 0 }
  }
  geometry Cylinder {
    radius 1
    height 3
  }
}
]
}
```

Da notare che essendo il campo rotation all' interno di un nodo Transform che si chiude prima della creazione del cilindro, la rotazione sarà applicata solo al parallelepipedo.

Il risultato a video:



6. Creazione di poligoni

Finora abbiamo visto la creazione di oggetti definibili tramite primitive pre esistenti nel dizionario VRML. Ma se vogliamo costruire un poligono o una figura solida non standard, abbiamo necessità di utilizzare un altro sistema.

La creazione di poligoni e di forme complesse è più agevole con un programma visuale, perciò vedremo qui soltanto un esempio didatico: la creazione di una piramide.

Il nodo preposto alla creazione di un poligono é **IndexedFaceSet** il quale ha due fields fondamentali: **coord** e **coordIndex**.

Il primo definisce i punti estremi delle facce del poligono, ossia i vertici, sotto forma di triplette di coordinate X, Y e Z di ogni punto.

Se vogliamo per esempio creare una piramide di 2 unità, la sezione coord sarà la seguente:

```
coord Coordinate {
        point [
              -2 0 2, 2 0 2, 2 0 -2, -2 0 -2, 0 2 0
           ]
```

Questo indica che il punto 0 della piramide avrà coordinate X=-2, Y=0, Z=2; il punto 1 avrà coordinate X=2, Y=0, Z=2 e così via.

Ragionandoci un po' sopra, i primi quattro punti rappresentano i 4 vertici della base (tutti con Y=0) e il quinto punto indica il vertice della piramide (Y=2 e X e Z= 0).

Il field coordIndex invece permette di decidere quali punti compongono ogni faccia.

Per esempio, la base sarà composta dai 2 vertici laterali davanti e dai 2 vertici laterali sul retro. Il field coordIndex non accetta coordinate ma la **posizione in cui le coordinate sono poste nel field coords** procedendo in senso antiorario.

Nella nostra definizione con coord abbiamo la tripleta -2 0 2 in posizione 0, la tripleta 2 0 2 in posizione 1 e così via.

Perciò per costruire la faccia della base, sia essa visibile o meno, dovremo attenerci alla tabella seguente:

-2 0 2	posizione 0 = angolo frontale sinistro
2 0 2	posizione 1 = angolo frontale destro
2 0 –2	posizione 2 = angolo retrostante destro
-2 0 –2	posizione 3 = angolo restrostante sinistro

dopo la dichiarazione di ogni faccia aggiungiamo il valore –1 per specificare che i vertici di definizione sono finiti.

Quindi il coordIndex per la base sarà: **coordIndex 0, 3, 2, 1, -1**

Se guardiamo la piramide di fronte, la faccia che vedremo sarà composta dal punto frontale sinistro, da quello frontale destro, e dal vertice alto, quindi la sua definizione coordIndex sarà: **coordIndex 0, 1, 4, -1**

Dopo aver ricavato le definizioni coordIndex di ogni singola faccia avremo il seguente risultato:

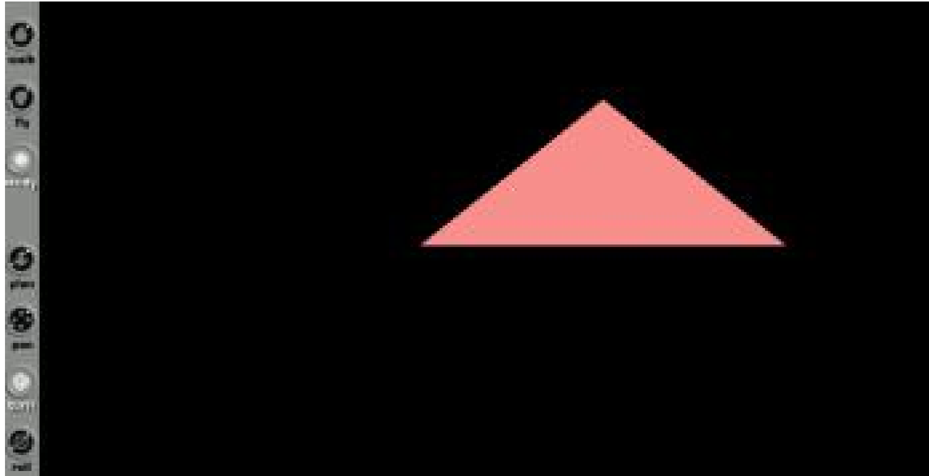
```
coordIndex [
        0, 3, 2, 1, -1
       0, 1, 4, -1
       1, 2, 4, -1
       2, 3, 4, -1
       3, 0, 4, -1
      ]
```

Possiamo ora scrivere il listato completo e commentarlo:

```
#VRML V2.0 utf8
#la piramide
   Shape {
     appearance Appearance {
      material Material { emissiveColor .8 0 0 }
     }
        geometry IndexedFaceSet {
         coord Coordinate {
       point [ -2 0 2, 2 0 2, 2 0 -2, -2 0 -2, 0 2 0 ]
         }
      coordIndex [
          0, 3, 2, 1, -1
       0, 1, 4, -1
       1, 2, 4, -1
       2, 3, 4, -1
       3, 0, 4, -1
      ]
     }
    }
```

il field geometry dichiara il nodo ‘IndexedFaceSet’ che indica che stiamo lavorando con un oggetto multifaccia, il suo primo field ‘coord’ dichiara un nodo ‘Coordinate’ il cui field ‘point’ contiene le coordinate dei 5 vertici. In ultimo, il field ‘coordIndex’ contiene la matrice che definisce i vertici di ogni faccia.

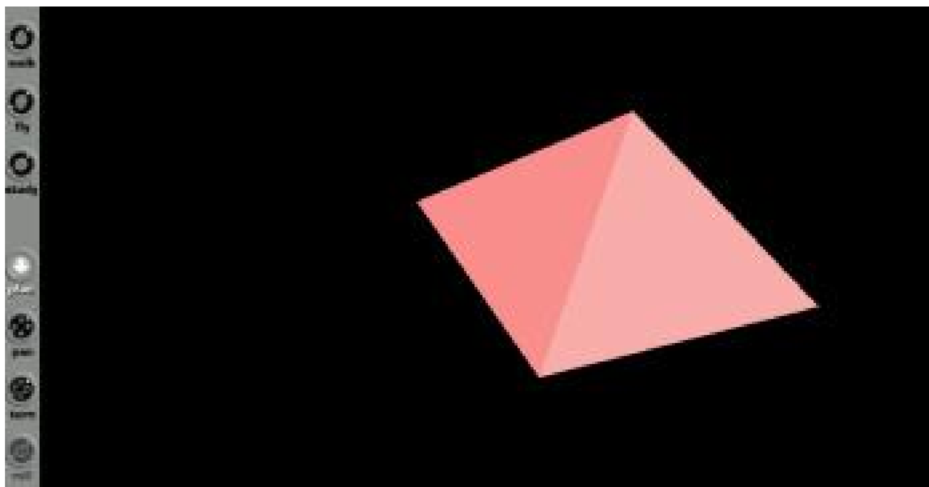
Il risultato a video:



Per poter vedere le varie facce della piramide potremo per esempio inserire la sua definizione in un transform che la ruoti lungo gli assi X e Y del tipo:

Transform {
  rotation 1 1 0 0.78
    children [

ricordandoci sempre, dopo la definizione di Shape, di chiudere sia il children che il Transform. Il risultato sarà una piramide ruotata di 45 gradi verso destra e verso il basso, come nell’ immagine seguente.



Se a questo punto vogliamo poggiare una sfera di raggio 2 sulla punta della piramide di altezza 2, prima di definire la sfera dovremo effetuare un Transorm di questo tipo:

```
Transform {
  translation 0 4 0
    children [
```

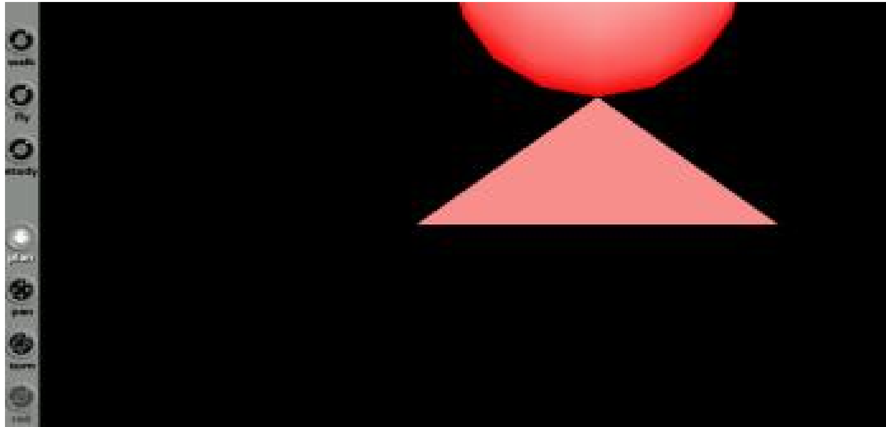
Dove il 4 è la somma delle 2 unità di raggio della sfera e delle 2 unità di elevazione positiva della piramide.

Il listato completo:

```
#VRML V2.0 utf8
#la piramide con sopra una palla
     Shape {
       appearance Appearance {
         material Material { emissiveColor .8 0 0 }
       }
            geometry IndexedFaceSet {
              coord Coordinate {
          point [
                   -2 0 2, 2 0 2, 2 0 -2, -2 0 -2, 0 2 0
                 ]
               }
         coordIndex [
                 0, 3, 2, 1, -1
          0, 1, 4, -1
          1, 2, 4, -1
          2, 3, 4, -1
          3, 0, 4, -1
         ]
       }
      }
Transform {
  translation 0 4 0
    children [
Shape {
  appearance Appearance {
    material Material { emissiveColor 1 0 0 }
```

```
  }
  geometry Sphere { radius 2 }
}
]
}
```

Il cui risultato sarà:



7. Accendiamo le luci

Gli oggetti che finora abbiamo creato sono semplici forme geometriche renderizzate che mancano di ogni qualità che possa conferirgli un effetto realistico.

Gli elementi principali utilizzabili per dare un realismo a un oggetto sono un rivestimento e una luce.

In un mondo VRML, se non accendiamo almeno una luce, non vedremo niente. Il fatto che noi vediamo gli oggetti creati senza bisogno di utilizzare luci è giustificato solo dall' utilizzo del campo **emissivecolor**. Questo fa sì che l' oggetto 'splenda' di una luce colorata che emana dalla superficie dell' oggetto stesso e **muore sulla superficie stessa**. Teniamo bene a mente questo dettaglio perché ci torneremo più avanti.

Per illuminare invece gli oggetti dall' esterno, abbiamo tre possibilità con VRML, i nodi:

- DirectionalLight
- SpotLight
- PointLight

Vediamoli ora in dettaglio:

7.1 Il nodo DirectionalLight

La sintassi del nodo DirectionalLight è esplicativa della sua funzione. Consideriamo la sua definizione:

```
DirectionalLight {
 direction 0 -1 -1
}
```

Come dice il termine, il field 'direction' definisce la direzione del fascio di luce, in questo caso che si propaga dall' alto verso il basso (Y=-1) e da davanti verso dietro (Z=-1).

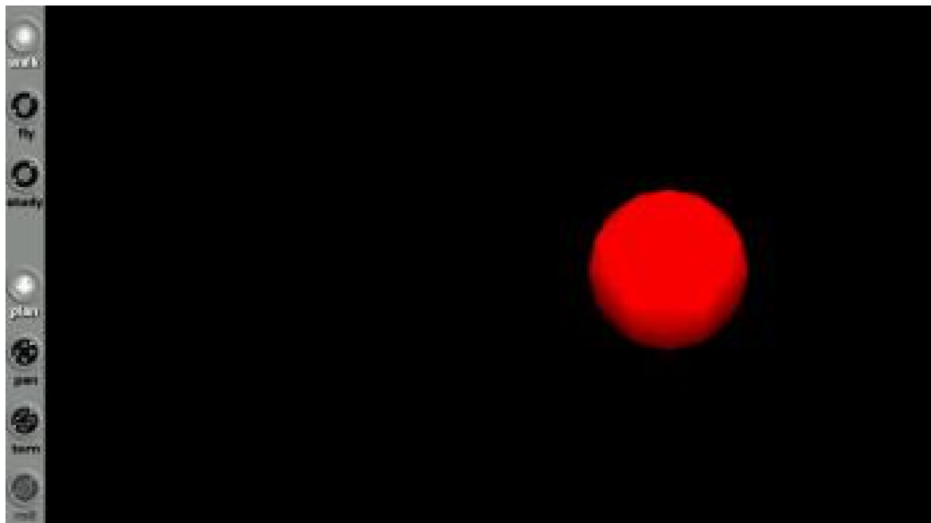
Scriviamo un listato che illumini una piccola sfera di raggio 2:

```
#VRML V2.0 utf8
DirectionalLight {
 direction 0 -1 -1
}
Shape {
 appearance Appearance {
  material Material { diffuseColor 1 0 0 }
```

```
  }
  geometry Sphere { radius 2 }
 }
```

notare che abbiamo sostituito emissiveColor con il nuovo campo **diffuseColor** che assegna un colore rosso non raggiante, in modo da non rendere vano l' utilizzo del fascio direzionale.

Salviamo in wrl e ammiriamo il risultato:



Come potrete notare, alla base della sfera si intravvede una zona in penombra dove la luce, proveniente dall' alto, non arriva.

Una caratteristica di DirectionalLight è infatti che crea una serie di raggi luminosi paralleli, e tutti gli oggetti della scena saranno illuminati dallo stesso fascio. La penombra o ombra dei singoli oggetti dipende solo dalla posizione che essi occupano rispetto al fascio.

Per illuminare diversi oggetti in modo diverso, dovremo utilizzare il prossimo nodo: SpotLight.

7.2 Il nodo SpotLight

Come spiegato poco fa, il nodo **SpotLight** serve per puntare una luce, una sorta di faretto virtuale, su un oggetto ben definito.

Al contrario di DirectionalLight, che va usato al di fuori della struttura children, il nodo SpotLight **va dichiarato all' interno**. Ciò rende possibile puntare un faretto con caratteristiche diverse su ogni oggetto.

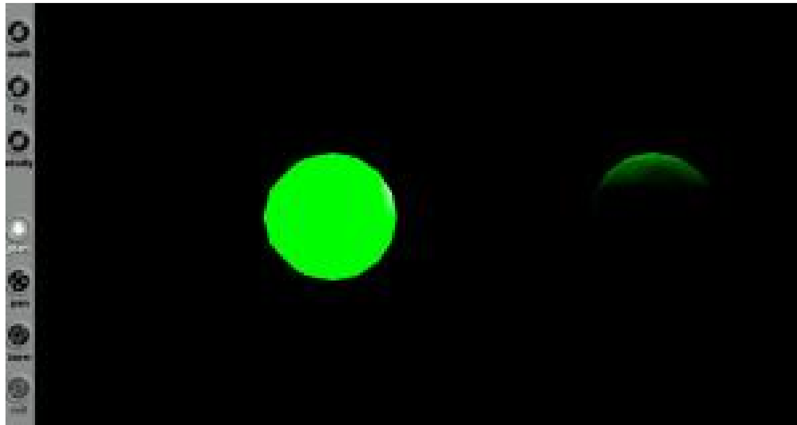
Vediamo subito un listato di esempio, che mostra una sfera con emissiveColor e una sfera con SpotLight:

```
#VRML V2.0 utf8
# sfera senza faretto e sfera con spotlight
Transform {
  translation -3 0 0
  children [
    Shape {
      appearance Appearance {
     material Material {
       emissiveColor 0 1 0
     }
      }
      geometry Sphere {  radius 1  }
    }
  ]
}
Transform {
  translation 2 0 0
  children [
    SpotLight {
      direction 0 -3 2
      location 0 3 -1
      radius 5
      beamWidth 1
      cutOffAngle 1
    }
    Shape {
      appearance Appearance {
        material Material {  diffuseColor 0 1 0  }
      }
      geometry Sphere {  radius 1  }
    }
  ]
}
```

Per evidenziare l' effetto, son state spente le **headlights** del browser VRML che a volte sfalsano l' effetto delle luci VRML. Con questo esempio è stato puntato sulla sfera di destra un

faretto a 3 unità in alto e 1 unità indietro rispetto all' origine, con una direzione che va di 3 unità in basso ma di 2 unità in avanti, per illuminare leggermente la parte di sfera che ci troviamo di fronte.

Il risultato a video:



Notare come anche emissiveColor ha un effetto diverso senza le headlights del browser.

Ora modifichiamo la creazione della sfera di sinistra in:

```
children [
  SpotLight {
    direction 0 1 -1
    location 0 -1 1
    radius 4
    beamWidth 1
    cutOffAngle 1
  }
  Shape {
    appearance Appearance {
  material Material {
    diffuseColor 0 1 0
  }
    }
    geometry Sphere {  radius 1  }
  }
 ]
```

puntando cioè un faretto davanti alla sfera e in una posizione di una unità più bassa, e angolandolo verso l' alto e verso la sfera stessa.

L' effetto che si ottiene è quello di illuminare la parte bassa della sfera.

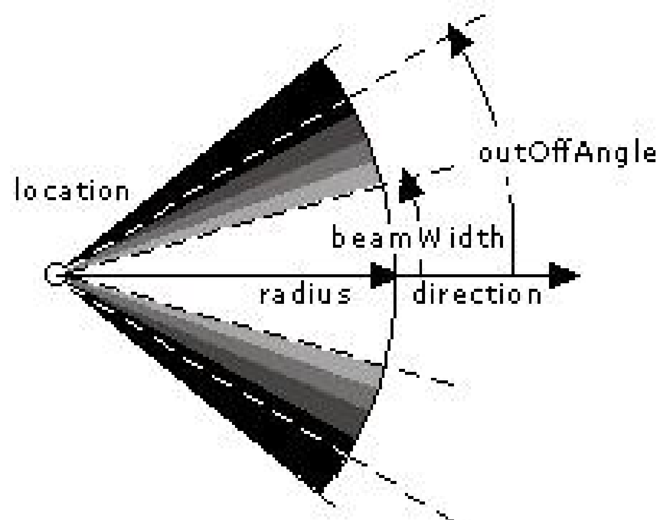
Ecco il risultato:

Notare che sulla destra, la sfera di sinistra presenta una illuminazione dovuta al faro posto dietro la sfera di destra.

Ciò dipende dal parametro '**radius**' del faretto che stabilisce di quanto si deve espandere il raggio al momento in cui 'tocca' l' oggetto verso cui è puntato.

I due parametri aggiuntivi, '**beamWidth**' e '**cutOffAngle**' servono per determinare l' 'alone' di luce e come esso si attenua.

L' immagine qui sotto è abbastanza esplicativa del significato di ogni parametro:



7.3 Il nodo PointLight

Il nodo PointLight è molto simile nei risultati al nodo SpotLight, anche se ha parametri leggermente diversi.

Viene utilizzato per far scaturire una luce che si estende per un raggio dichiarato dal parametro **radius** (n) da una sorgente di cui specifichiamo la posizione tramite il parametro **location** (x y z).

PointLight permette di ottenere un effetto di attenuazione, ma generalmente i suoi effetti visivi sono poco apprezzabili specialmente se, come negli esempi precedenti visti per SpotLight, si tengono 'accese' le Headlights.

Scriviamo questo script:

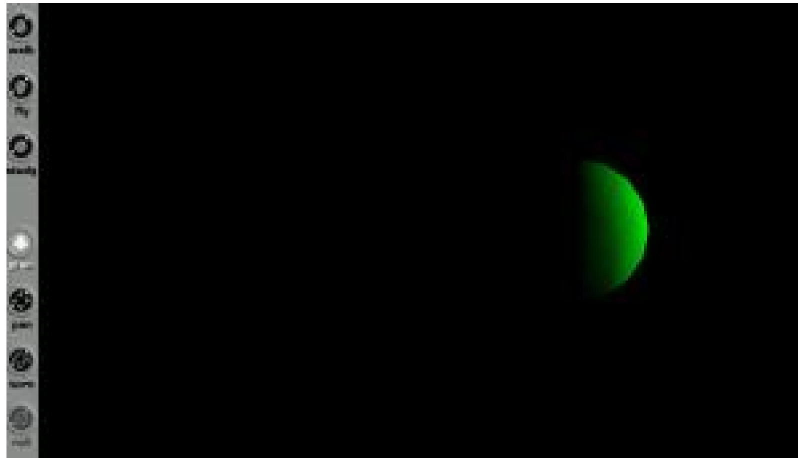
```
#VRML V2.0 utf8
# sfera con pointLight

PointLight {
  radius 10
  location 6 1 0
}
    Shape {
      appearance Appearance {
    material Material {
      diffuseColor 0 1 0
    }
      }
      geometry Sphere {  radius 1  }
    }
```

come vedete è molto semplice.

Abbiamo soltanto sistemato una luce '*puntuale*' sullo stesso piano della sfera ma spostato di 4 unità verso destra.

Il risultato sarà molto simile all' illuminazione già ottenuta con SpotLight, come si può vedere nell' immagine seguente:



Non c’ è molto altro da dire sull’ illuminazione degli oggetti nei mondi VRML.

Ora possiamo riaccendere le luci headlights del browser e dedicarci a dare un po’ più di realtà ai nostri oggetti.

Il prossimo passo sono le textures.

8. Le textures

Finora abbiamo creato solo oggetti colorati, il chè in un mondo VRML può essere utile ma fino ad un certo punto. Per il concetto stesso e per il significato del nome di questo linguaggio (Linguaggio per la modellazione di **Realtà** Virtuale) sarebbe estremamente limitativo fermarsi a utilizzare colori.

La realtà è quasi sempre composta da motivi, più che da colori. Pensiamo per esempio a un mattone. Ha una forma parallelepipeidale, ma non ha un colore uniforme. O una mattonella marmorizzata, o a una panca di legno.

Non potremo mai ottenere l' effetto legno con i colori. E' a questo che servono le textures.

Prendiamo il solito esempio della sfera: e se volessimo trasformarla in una biglia marmorizzata?

VRML ci mette a disposizione il field '**texture**' e il nodo **ImageTexture** per aggiungere trame e motivi ai nostri oggetti.

Prendiamo il nostro primo programmino riguardante la sfera e modifichiamolo come segue:

```
#VRML V2.0 utf8
# Sfera di marmo

Shape {
  appearance Appearance {
    material Material { }
    texture ImageTexture {url "marmo.jpg"}
  }
  geometry Sphere { radius 1 }
}
```

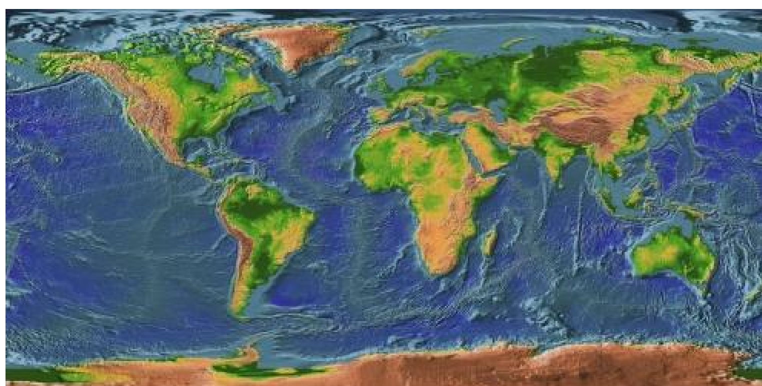
Ovviamente, dovremo avere il file marmo.jpg nella stessa cartella dove è situato il mondo VRML. La sintassi del comando utilizzato è facilissima, non ha bisogno di spiegazioni, quindi evidenziamo semplicemente che nell' esempio è stato eliminato ogni comando di colorazione (nel field Material abbiamo tolto l' emissiveColor). Questa è una prassi spesso poco conveniente perché a seconda dell' ambiente, non dichiarando nessun colore potremmo avere l' inconveniente di non veder visualizzato l' oggetto in caso il file di texture non fosse presente.

Lanciamo lo script e vediamo il risultato a video:

Tutt’ altra cosa vero?

Una tecnica per evitare il problema precedentemente esposto, sta nel dare comunque all’ oggetto un diffuseColor o un emissiveColor il più simile possibile al colore dominante nella texture, in questo caso per esempio avremmo potuto cercare il bianco o un azzurrino pallido.

Divertiamoci un po’ a creare un mini mappamondo che rappresenti una cartina fisica del nostro pianeta. Ovviamente dovremo procurarci una immagine di questo genere:



Una volta trovata la texture la applichiamo alla nostra solita sfera:

```
#VRML V2.0 utf8
# Pianeta terra

Shape {
  appearance Appearance {
    material Material { }
    texture ImageTexture {url "terra.jpg"}
  }
  geometry Sphere { radius 1 }
}
```

Ed ecco il nostro planisfero:

Ovviamente il nodo ImageTexture si occuperà automaticamente di ridimensionare l' immagine texture per farla combaciare alle dimensioni dell' oggetto VRML.

Un ultima parola riguardo alle textures: VRML permette di utilizzare Gif trasparenti come textures, e anche filmati in formato MPEG-1 per creare textures animate.

In questo caso, il nodo da utilizzare sarà **MovieTextures**, e la sua dichiarazione sarà:

texture MovieTexture {
url "movie.mpg"
loop trUE
}

il parametro '**loop**' stabilisce se il filmato va ripetuto in ciclo (true).

Se lo si vuole far eseguire solo una volta basta omettere il field 'loop'.

Una nota: se volessimo utilizzare una superficie a più facce, per esempio un cubo, ed avere su ogni faccia una texture diversa, non dovremmo usare una primitiva come Box, ma costruire un poligono e dichiarare ogni singola faccia usando il metodo già visto che prevede il nodo 'IndexedFaceSet' e i fields 'coord' e 'coordIndex'. Ma quasi mai queste operazioni si effettuano scrivendo codice a mano. Sono operazioni molto più veloci e sicure da svolgere con un editor visuale.

Vediamo ora come creare uno sfondo, per eliminare il senso di 'vuoto' dei mondi creati finora.

9. Uso avanzato delle Textures: creazione di sfondi, poligoni mappati e oggetti complessi.

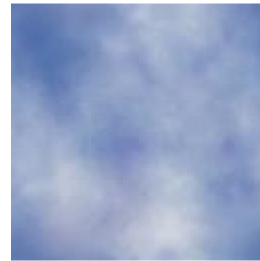
9.1 Sfondi con i nodi

Per creare un ambiente intorno a noi, esistono due metodi.

Usare una serie di textures, o delle gradazioni di colori. In questo manuale analizzeremo solo la prima possibilità. Dovremo munirci perciò di textures che rappresentino l' ambiente che ci deve circondare.

L' esempio più semplice, se supponiamo di collocare il nostro mondo all' aperto, potrebbe essere quello di trovare due textures del cielo e di un prato o di un tipo di pavimentazione.

Come ad esempi queste:



VRML mette a disposizione il nodo '**Background**' nel quale dichiarare i seguenti campi:

- backUrl
- bottomUrl
- frontUrl
- leftUrl
- rightUrl
- topUrl

ognuno con la funzione di 'coprire' una porzione di spazio.

Modifichiamo il listato dell planisfero aggiungendo i parametri per lo sfondo:

```
#VRML V2.0 utf8
# Pianeta terra con sfondo

Background {

backUrl [ "clouds.gif" ]
```

```
bottomUrl [ "ground.gif" ]
frontUrl [ "clouds.gif" ]
leftUrl [ "clouds.gif" ]
rightUrl [ "clouds.gif" ]
topUrl [ "clouds.gif" ]

}
Shape {
  appearance Appearance {
    material Material { }
    texture ImageTexture {url "terra.jpg"}
  }
  geometry Sphere { radius 2 }
}
```

Il risultato come potete vedere non è il massimo:

Questa è l' immagine che ritrae lo sfondo del cielo. Mentre se con gli strumenti di navigazione angolate la visuale verso il basso vedrete sbucare il pavimento:

9.2 Sfondi e ambienti con poligoni mappati.

Se esplorate un po’ questo mondo appena creato vedrete che il ‘pavimento’ è mal delimitato. Questo perché comunque si tratta di uno spazio aperto che si estende teoricamente all’ infinito.

Per ottenere qualcosa di migliore, come scritto prima, bisognerebbe costruire un mega poligono molto ampio che definisca 6 facce:

- il cielo
- il pavimento
- le 2 pareti frontali
- le 2 pareti laterali

Giusto a scopo di esempio riporto qui lo script per creare un pavimento quadrato di 20 unità x 20 unità:

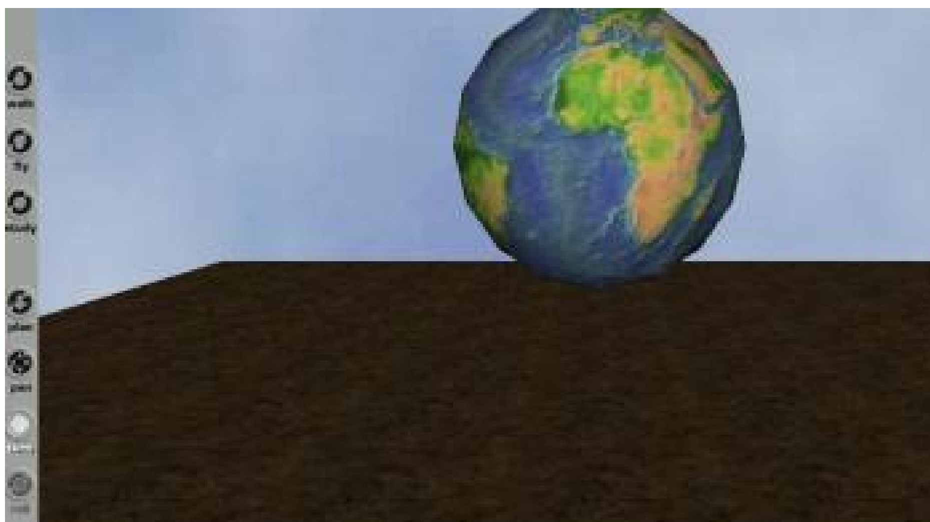
```
#VRML V2.0 utf8
# Pianeta terra con sfondo poligono

Background {
backUrl [ "clouds.gif" ]
bottomUrl [ "ground.gif" ]
frontUrl [ "clouds.gif" ]
leftUrl [ "clouds.gif" ]
rightUrl [ "clouds.gif" ]
}
Shape {
  appearance Appearance {
    material Material { }
    texture ImageTexture {url "terra.jpg"}
  }
  geometry Sphere { radius 2 }
}
Transform {
translation 0 -3 0
children [
Shape {
          appearance Appearance {
            material Material { diffuseColor 1 1 0 }
    textureTransform TextureTransform { scale 10 10 }
```

```
texture ImageTexture {url "ground.gif"}
  }
        geometry IndexedFaceSet {
          coord Coordinate {
      point [
                -10 0 10, 10 0 10, 10 0 -10, -10 0 -10
             ]
          }
    coordIndex [
            0, 1, 2, 3, -1
      ]
  }
 }
]
}
```

Si noti che è stato eliminato il field 'bottomUrl' dal nodo 'Background' altrimenti avremmo avuto un doppio pavimento.

Il risultato:



Appena si carica il mondo e l' immagine è frontale il pavimento appare nero, perché non è illuminato da nessuna luce, ma angolando un po' verso il basso, come potete vedere, si inizia a vedere la texture. Un' altra cosa da notare sul listato è l' utilizzo di:

textureTransform TextureTransform { scale 10 10 }

questa procedura viene usata per '*scalare*' una texture, ossia per ripetere più volte su una superficie la stessa texture, per esempio per ottenere maggior dettaglio. In questo caso è stato essenziale perché altrimenti la texture del pavimento sarebbe stata 'espansa' per apparire una sola volta su tutta la faccia definita di 10x10 unità ottenendo un livello di dettaglio bassissimo.

9.3 Creazione di oggetti complessi: l' albero

Il nodo '**Billboard**', che ora vedremo in azione, ha la caratteristica di poter raggruppare vari oggetti all' interno della sua struttura children. Questo può tornare utile quando si vuole comporre un oggetto con varie forme alle quali applicare le stesse procedure.

In questo listato seguente utilizziamo il nodo Billboard su un oggetto soltanto, ma potremmo utilizzarlo su gruppi di oggetti.

Lavoriamo sempre con textures, e stavolta con una gif trasparente di un albero.

Il listato è il seguente:

```
#VRML V2.0 utf8
#esempio di utilizzo del nodo Billboard – l' albero
Billboard {
axisOfRotation 0 1 0
children [
Shape {
appearance Appearance {
material Material { diffuseColor 1 1 1 }
texture ImageTexture {
url "tree.gif"
}
}
geometry Box { size 3 4 .1 }
}
]
}
Transform {
translation 0 -2 0
children [
Shape {
	appearance Appearance {
```

```
material Material { diffuseColor 1 1 0 }
textureTransform TextureTransform { scale 10 10 }
texture ImageTexture {url "ground.gif"}
}
geometry IndexedFaceSet {
coord Coordinate {
point [
-10 0 10, 10 0 10, 10 0 -15, -10 0 -15
]
}
coordIndex [
0, 1, 2, 3, -1
]
}
}
]
}
```

Angolando opportunamente la scena perché si veda anche il pavimento, il risultato è questo:

Realistico vero?

Il trucco in questa creazione è aver usato un box con profondità praticamente nulla. In questo modo evitiamo che, guardando l' albero dall' alto, si veda di nuovo la trama sulla faccia superiore del box.

Nella struttura:

Billboard {
axisOfRotation 0 1 0
children [

axisOfRotation definisce l' asse lungo la quale deve essere calcolata la rotazione delle varie primitive contenute dentro la struttura children. In questo caso poco influente inquando abbiamo un solo oggetto.

10. Collisioni: costruzione di oggetti solidi

Se aprite uno qualsiasi degli esempi finora creati, e vi spingete con i tasti di navigazione fino agli oggetti, noterete che riuscite a 'passare attraverso'. Questo, ovviamente, in un mondo virtuale degno di essere chiamato tale è un fenomeno che deve essere controllato. Come faremmo altrimenti a costruire una casa a più stanze? Potremmo passare attraverso i muri rendendo inutile la creazione di porte, finestre, e rendendo comunque il nostro mondo lontanissimo dalla realtà simulata.

Ci viene incontro il nodo **Collision** il cui comportamento e la cui sintassi sono semplicissimi. Infatti basta creare gli oggetti come children del nodo Collision perché questi oggetti abbiano caratteristica di solidità.

Il classico esempio della pallina, rielaborato come oggetto children del nodo Collision, diventa così:

```
#VRML V2.0 utf8
# semplice esempio di collision detection su sfera

Collision {
  children [
    Shape {
      appearance Appearance {
        material Material { emissiveColor 1 0 0 }
      }
      geometry Sphere { radius 2 }
    }
  ]
}
```

Non si può rendere un effetto a video del risultato collision, ma vi invito a provare l' esempio cercando di attraversare la sfera per capirne il funzionamento.