

VRML - A First Look

Thomas Larsson

Department of Computer Engineering

Mälardalen University

February 7, 1999

Västerås

Sweden

Abstract

This tutorial gives an introduction to the Virtual Reality Modeling Language (VRML). First of all, the basics of how to build virtual worlds using this language are explained. Then follows a brief presentation of how these worlds can be optimized to achieve better rendering performance and, finally, some techniques of how to bring these virtual worlds alive are exemplified.

Contents

1. Introduction	4
2. Creating Basic Virtual Worlds in VRML.....	4
2.1 Building a Very Simple World	4
2.2 Using Transformations	8
2.3 Defining More Interesting Geometry.....	11
2.4 Reusing Nodes	12
2.5 Lighting.....	14
2.6 Basic Texture Mapping	17
2.7 Controlling the Viewer	22
3. Optimization Techniques.....	23
3.1 Adaptive Level of Detail	24
3.2 Billboards	25
4. Moving Beyond Static Worlds.....	27
4.1 Allowing User Interaction	27
4.2 Introducing Scripts	29
5. VRML on the Internet	31
5.1 Adding Hyperlinks	31
5.2 Combining VRML and HTML on Web Pages.....	32
6. Conclusions	33
7. Acknowledgements.....	33
References	33

1. Introduction

VRML, short for Virtual Reality Modeling Language, is often pronounced ‘vermal’. This language was created to add 3D graphics possibilities to the World Wide Web. So far, however, the acceptance of VRML in the Web community hasn’t exactly been overwhelming, but with the constantly increasing performance available in new PC’s and people getting used to the Virtual Reality concept, we might see a change of that in the near future.

In 1997 VRML97¹ was accepted as an ISO standard [Vrm97]. It is quite an improvement compared to VRML 1.0 from 1994. Some parts have been changed and many new features have been added, especially the interactive capabilities have been greatly improved. In this tutorial we will cover VRML97, which still is the latest version, but since this is meant to be a short introductory tutorial a lot of interesting topics will be left out. There are many textbooks that can fill in the gaps and guide the interested reader much further.

To fully understand the material presented throughout the rest of this tutorial a general knowledge of 3D computer graphics is assumed. In particular, knowing the theory behind geometrical transformations, viewing, lighting calculations and texture mapping will be helpful to understand what’s going on.

The rest of this tutorial is organized in the following way. After describing how to create basic three-dimensional virtual worlds, we will explore some possibilities of how to optimize these worlds. Finally, we will deal with possible ways of how to let users interact with the virtual environment.

2. Creating Basic Virtual Worlds in VRML

Worlds in VRML are described using the VRML file format. Since this format is text based it is possible to use some general text editor and type in our world descriptions. When naming our files, we use the .wrl extension. To explore a world, described in a VRML file, we can for example use the free VRML browser *Cosmo Player* together with *Netscape Communicator*. All the worlds in this tutorial have been tried out using these mentioned programs², but of course there are other alternatives. Now, let’s turn our attention to our first VRML world.

2.1 Building a Very Simple World

To build a virtual world in VRML we need to specify different *nodes*, representing certain geometry, qualities or actions, and their relationships. In this way, we define the structure of the world, also known as the scene graph. To see how this works we can look at a very simple world, consisting of only a cylinder object and a cone object, both centered at the origin in the world:

¹ VRML97 is the ISO standard revision of VRML 2.0 from 1996

² To be more exact, Cosmo Player 2.1 and Netscape Communicator 4.5 have been used.

Listing 1

```
#VRML V2.0 utf8
#A very simple VRML world

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1 1 0.5
    }
  }
  geometry Cylinder { }
}

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1 0 0.5
      specularColor 1 1 1
      shininess 0.1
    }
  }
  geometry Cone {
    bottomRadius 0.8
    height 4
  }
}
```

Before we discuss the meaning of this VRML code in detail, take a look at how *Cosmo Player* initially render this simple world in Figure 1. Having this picture in mind, let's examine the code used to describe this simple world.

First of all, note that a VRML file have to start with a VRML header line:

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

It indicates the file is a VRML file, version 2, and in the following text UTF8 encoding is used. UTF8 is an ISO standard making the representation of characters from many different languages possible. After the header line, a comment follows. All comments start with the character # and continues to the end of the line. Comments are ignored by VRML browsers, but of course they are a good tool for authors to keep track of different sections in a file.

The Shape node is used to associate the actual geometry of an object with its appearance, which is information controlling the look of the object. The definition of the Shape node looks like this:

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

Every node in VRML consist of a list of fields that holds important values for its functionality. The definition of a node shows us all these fields and gives us quite a lot of information of how that node can be used. For example, in the definition of the Shape node, there are four columns inside the node. In the first column we find the *class specifiers*. The specifier used in this case, `exposedField`, serves many purposes. Among other things an `exposedField` can be changed as a result of an event and they are always accessible to scripts. In the second column, we find the data type of the fields. In the Shape node, the data type for both fields are `SFNode`, which is a data type referring to other nodes. Hence, a Shape node is used to group two other nodes together. The last two columns describe the parts you actually can type into a VRML file. There you find the names of the fields together with suitable default values, used when they are not specified.



Figure 1: The world in listing 1 as rendered by Cosmo Player

In the `Appearance` node, we can specify what material, e.g. color, or what texture, will be used for the associated geometry. Its definition looks like this:

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

This node is also used to hold other nodes. The `material` field can refer to a `Material` node, which describes color information for surfaces. The `texture` and `textureTransform` fields refers to nodes making different types of texture mapping

possible. In our simple example world, we only used the `material` field. The corresponding node is defined as follows:

```
Material {
    exposedField    SFFloat  ambientIntensity    0.2
    exposedField    SFColor   diffuseColor     0.8 0.8 0.8
    exposedField    SFColor   emissiveColor      0 0 0
    exposedField    SFFloat   shininess         0.2
    exposedField    SFColor   specularColor     0 0 0
    exposedField    SFFloat   transparency      0
}
```

Since we didn't specify all these fields in our example, many of the default values from this definition are used. For example the `ambientIntensity` field, that can hold a value between 0 and 1 to specify the amount of background light present, take its default value 0.2 in our case. To specify colors in VRML the RGB color model is used. Thus, the basic color of the material, held by the `diffuseColor` field, are specified using three values between 0 and 1 for each color component red, green and blue respectively. In the same way, the `specularColor` and the `emissiveColor` of the material are specified. The `specularColor` field controls the color of highlights on shiny objects and it is used together with the `shininess` field to control the size of the highlights. If you carefully examine the look of our example world, you will notice that it is only possible to get a highlight on the cone object. When it comes to the cylinder object, no `specularColor` and no `shininess` value were given and, hence, no highlight is possible in that case. The `transparency` field makes it possible to simulate transparent surfaces. A transparency value of 0 means the material is totally opaque and a value of 1 means it is totally transparent.

The material information discussed here is used together with light source specifications to illuminate objects in the virtual world. So far we haven't defined any light sources, but still our world looks alright. The reason for this is that there is always a default light source used, at least in *Cosmo Player*, if no one is specified. Later, we will see how to specify our own light sources.

Finally, we conclude this section by looking at the geometry nodes `Cylinder` and `Cone`, also used in our example. These nodes are simple to specify and they can be rendered very efficiently. Here is the definition of the `Cylinder` node:

```
Cylinder {
    field    SFBool   bottom    TRUE
    field    SFFloat  height    2
    field    SFFloat  radius    1
    field    SFBool   side      TRUE
    field    SFBool   top       TRUE
}
```

As you can see, the meaning of these fields are easy to guess. A cylinder is given by its height and radius and you can choose what parts of it will be visible. And the `Cone` node is defined in exactly the same style:

```

Cone {
  field  SFFloat bottomRadius  1
  field  SFFloat height        2
  field  SFBool  side           TRUE
  field  SFBool  bottom        TRUE
}

```

Apart from these two primitive objects, there are two other primitives, `Box` and `Sphere`, supported by VRML as well. They are used in pretty much the same way and some of the following examples will show them in action.

When viewing our example world, you can't miss that these two primitives, the cylinder and the cone, are intersecting each other. That is because they are both centered around the world coordinate systems origin. These objects can easily be moved to other positions in the world with the help of geometrical transformations. We will see how to do that in the next section.

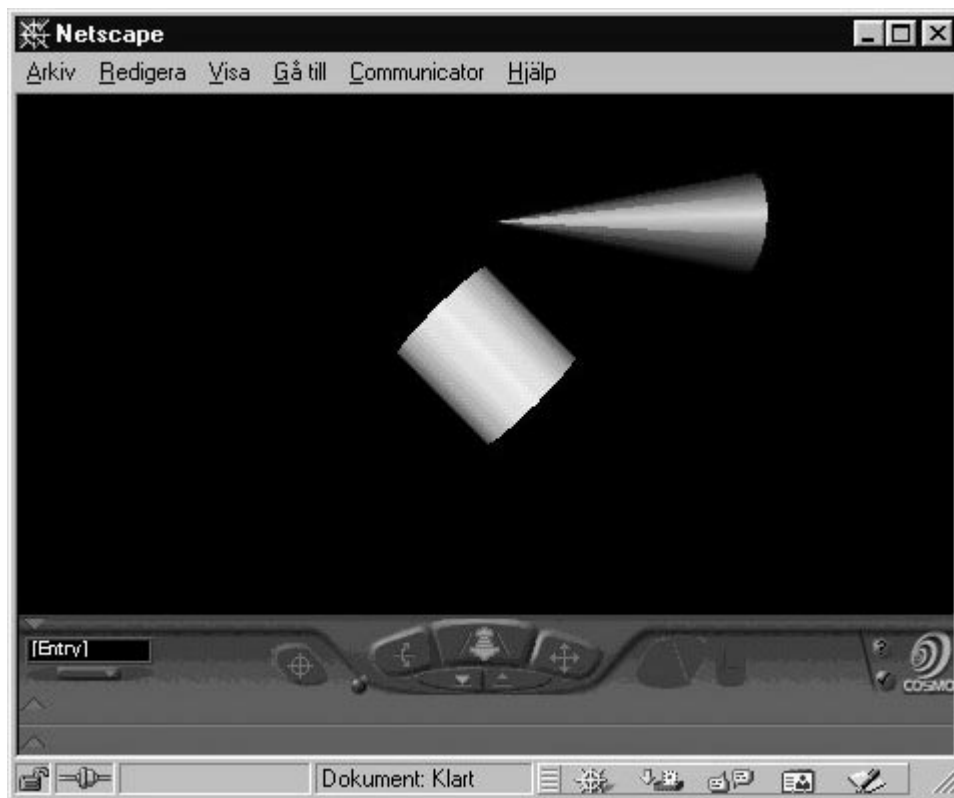


Figure 2: Transformed objects

2.2 Using Transformations

In our first example, we saw a world where two objects did intersect each other at the origin of the world coordinate system. That was because they were both placed there by the VRML browser. Obviously, we need a mechanism to avoid this situation. Such a mechanism is given to us by the `Transform` node, whose purpose is to make it possible to move geometrical objects around in the scene, as well as rotate and scale them. Here is an example showing how

we can change the position and orientation of the cylinder and sphere object we used before. The result can be seen in Figure 2.

Listing 2

```
#VRML V2.0 utf8
#An example illustrating geometrical transformations

Transform {
  rotation 0 0 1 0.78
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 1 1 0.5
        }
      }
      geometry Cylinder { }
    }
    Transform {
      translation 3 0 0
      rotation 0 0 1 0.78
      children Shape {
        appearance Appearance {
          material Material {
            diffuseColor 1 0 0.5
            specularColor 1 1 1
            shininess 0.1
          }
        }
        geometry Cone {
          bottomRadius 0.8
          height 4
        }
      }
    }
  ]
}
```

Note that the children field group a list of other nodes that will be affected by the transform specified. For example, it is possible to add both Shape nodes and other Transform nodes to this list to build up some useful scene graph. This make it possible to specify relative transformations between different objects. Here follows the complete definition of the Transform node:

```
Transform {
  eventIn          MFNode      addChilden
  eventIn          MFNode      removeChildren
  exposedField     SFVec3f     center          0 0 0
  exposedField     MFNode      children          []
  exposedField     SFRotation  rotation        0 0 1 0
```

```

    exposedField SFVec3f    scale      1 1 1
    exposedField SFRotation scaleOrientation 0 0 1 0
    exposedField SFVec3f    translation 0 0 0
    field SFVec3f    bboxCenter 0 0 0
    field SFVec3f    bboxSize   -1 -1 -1
}

```

The `translation` field simply holds the x-, y- and z-component respectively of a position in the world where to the children of the node will be moved (from their default position) and the `scale` field holds the amount of scaling that will be applied relative the three coordinate axes. But how do we specify the `rotation` field? We use four values, the first three describes a vector which defines the axis of rotation and the last value gives the rotation angle in radians around this axis. The `center` field allows us to specify a position about which the rotation occurs.

Another, more advanced scaling operation can be accomplished by using the `scaleOrientation` field, which allows us to rotate the axes, around which scaling occurs. This field is specified in the same way as the rotation field, that is, by first defining an axis followed by the rotation angle in radians. Finally, the two last fields, `bboxCenter` and `bboxSize`, are used to specify the position and size of a bounding box surrounding all of the children referred to by the node. If this information is not given, the VRML browser has to calculate them, which can be quite time consuming in some cases.



Figure 3: A pair of red triangle polygons

2.3 Defining More Interesting Geometry

Although it is possible to combine primitive objects such as boxes, cones, cylinders and spheres into much more interesting and complex objects, it is certainly not enough for most applications. What we need is a way to define the geometry of real world objects in a much more realistic manner, yet effective enough for the virtual reality concept we are targeting. The `IndexedFaceSet` node let us define arbitrarily patches built of flat convex polygons. By using this node we can create approximations of almost all kinds of objects. To show how to use this node, we will first consider a very simple case; how to put two red triangles in a VRML world:

Listing 3

```
#VRML V2.0 utf8
#Example: How to use the IndexedFaceSet node

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

As we can see, the `coord` field holds a `Coordinate` node, which defines the position of the five vertexes in the `IndexedFaceSet`. To specify which vertex belonging to which face we use the `coordIndex` field. For example, the first face is made up of vertex number 0, 1 and 2, in exactly that order, and the face is terminated by the integer `-1`. The two faces are visible from the beginning, when we view this world in *Cosmo Player*, as we can see in Figure 3. That's because their vertexes are facing the viewer in counter clockwise order. If we would go around these polygons and look at them from the backside, we would see nothing. We have to think about this when we create objects using this node. Here is the complete definition of the `IndexedFaceSet` node:

```
IndexedFaceSet {
  eventIn          MFInt32      set_colorIndex
  eventIn          MFInt32      set_coordIndex
  eventIn          MFInt32      set_normalIndex
  eventIn          MFInt32      set_textCoordIndex
  exposedField     SFNode       color          NULL
  exposedField     SFNode       coord          NULL
  exposedField     SFNode       normal         NULL
  exposedField     SFNode       texCoord      NULL
  field            SFBool       ccw           TRUE
```

```

    field          MFInt32      colorIndex      [ ]
    field          SFFloat      colorPerVertex  TRUE
    field          SFFloat      convex              TRUE
    field          MFInt32      coordIndex       [ ]
    field          SFFloat      creaseAngle          0
    field          MFInt32      normalIndex        [ ]
    field          SFFloat      normalPerVertex    TRUE
    field          SFFloat      solid                TRUE
    field          MFInt32      texCoordIndex     [ ]
}

```

There are quite many fields in this node. For now, we will just briefly explain what some of these fields mean. We have already seen how to use the `coord` field as well as the `coordIndex` field, but we can also assign normals to corresponding vertexes. To do so, we use the `normal` and `normalIndex` field.

The `convex` field defaults to `TRUE`, which means we can only use convex polygons in the node, but this requirement can be removed by setting this flag to `FALSE`. When doing so, non-convex faces can be used and VRML browsers are expected to treat these surfaces in an appropriate manner. The `ccw` field tells which side of the polygons are going to be rendered. As discussed before, it depends of the vertex ordering used in the descriptions of the polygons. You can choose which side of polygons that are going to be considered visible by changing this field. If both sides are going to be visible you simply set the `solid` field to `FALSE`.

Other geometry nodes in VRML are for example `IndexedLineSet`, `PointSet` and `ElevationGrid`. The last one mentioned is very useful for describing the ground by using a uniform grid with height values for each point in the grid. However, none of these nodes are discussed further in this tutorial.

2.4 Reusing Nodes

In a typical scene, much of the same information is used over and over again. For example, the same material specification might be needed for many objects or some objects might be needed in many copies. But even if we need ten instances of an object in a scene, we might only want to describe its shape once. For this reason, and also for several other reasons, there is a naming facility in the VRML file format. To define a name we put the `DEF` keyword followed by the name in front of a node. Then we can refer to this node later by putting the `USE` keyword, followed by the defined name, in place of a node. The following example shows us how it works:

Listing 4

```

#VRML V2.0 utf8
#Example: How to use the DEF and USE modifiers/keywords

DEF mySphere Shape {
    appearance DEF myApp Appearance {

```

```

        material Material {
            diffuseColor 1 0 1
        }
    }
    geometry Sphere { }
}

Transform {
    translation 0 -2 0
    children USE mySphere
}

Transform {
    translation 0 2 0
    children Shape {
        appearance USE myApp
        geometry Cylinder { }
    }
}

```

In this example, both a Shape node and an Appearance node are reused and the result is shown in Figure 4.

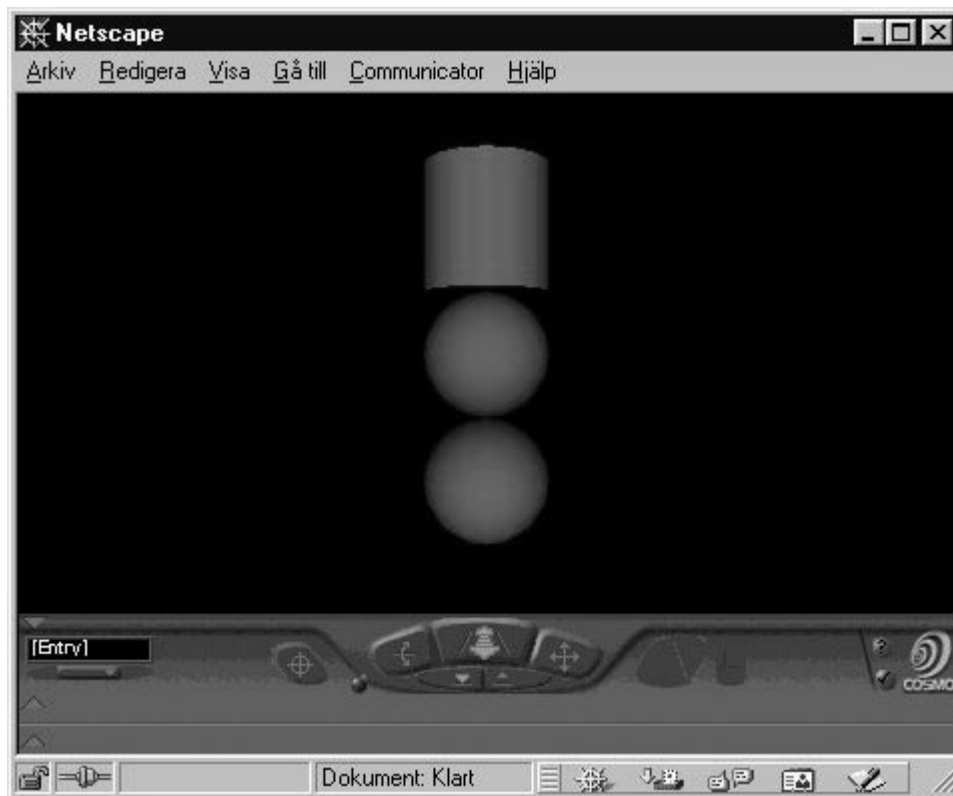


Figure 4: Reusing nodes in the scene graph

This reuse obviously makes the file shorter, which saves download time and loading time, but that's not the whole story. When the VRML parser finds a USE statement, a new copy of the referred node will not be created. Instead, a pointer will be established to refer to the original

node, marked by the corresponding DEF statement. In many cases that doesn't matter, or is even desirable, but remember; whatever happens to the original node also happens to the 'copies'. To avoid this effect in a certain case, simply avoid using the DEF/USE construction.

2.5 Lighting

One very important aspect of how to make a virtual world look nice and realistic is the specification of light sources. So far, we have only specified certain parameters controlling the appearance at object level. The reason why our examples have looked alright until now is that a light source have been added to our scenes automatically. However, we need to be able to specify our own light sources to be used in the lighting calculations together with the appearance information applied to the shapes in the scene. There are three different kinds of light sources in VRML, represented by the nodes `DirectionalLight`, `PointLight` and `SpotLight`. The simplest one is the `DirectionalLight`. Let's see an example of how to use such a light source:

Listing 5

```
#VRML V2.0 utf8
#Example: Using one global directional light

DirectionalLight {
  direction 1 0 0
}

DEF myCone Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.7 0.7 0.1
    }
  }
  geometry Cone { }
}

Transform {
  translation -3 0 0
  children USE myCone
}

Transform {
  translation 3 0 0
  children USE myCone
}
```

All we have to do to add a directional light source to a scene is to enter the direction of the parallel light rays, which is given by a vector. Note that the directional light source doesn't need to have any given position in the world. Instead, it is used to simulate light sources infinitely far away, like the sun, for instance.

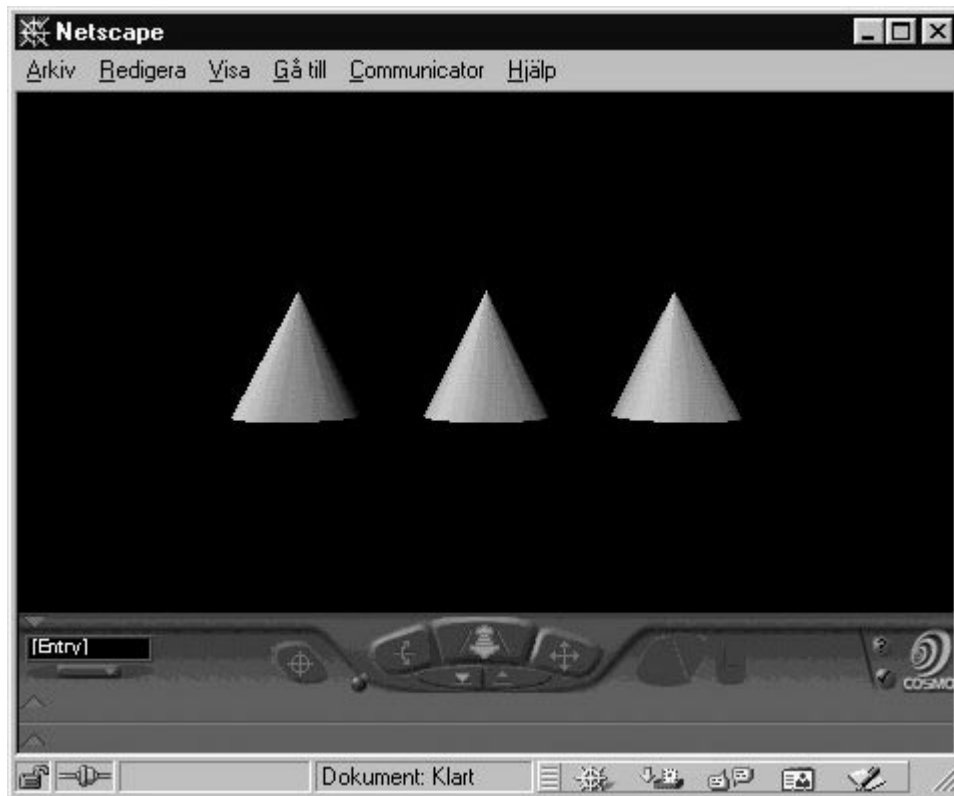


Figure 5: Using a global directional light source

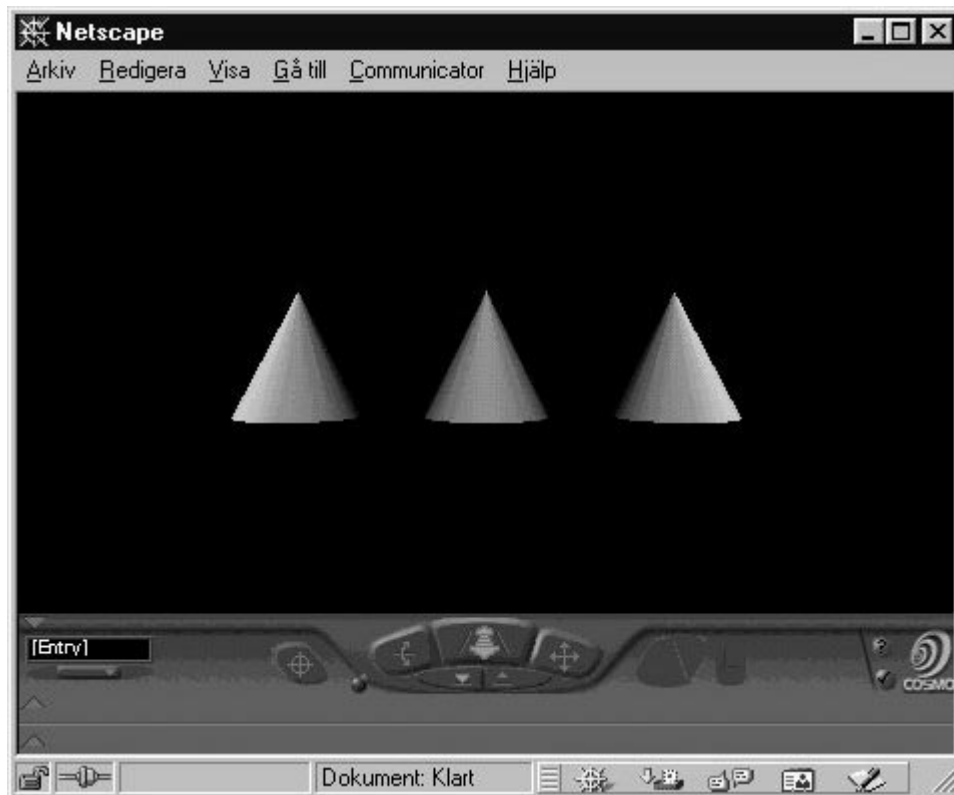


Figure 6: Using two scoped directional light sources

The following definition of the `DirectionalLight` node shows us the options available to control this kind of light source:

```
DirectionalLight {
  exposedField SFFloat    ambientIntensity 0
  exposedField SFColor    color          1 1 1
  exposedField SFVec3D    direction        0 0 -1
  exposedField SFFloat    intensity        1
  exposedField SFBool     on                TRUE
}
```

Here we can see that the `color` field allows us to choose an RGB color for the light rays and if we don't, the light defaults to white. We can also adjust the overall intensity of the light as well as turning the light on and off using the `on` flag. And if we set the `ambientIntensity` field it will be added to the `ambientIntensity` field in the current `Material` node throughout the lighting calculations.

Another interesting aspect of the `DirectionalLight` node is that it will only affect nodes placed under it in the scene graph. The light source in the example above is global to the whole scene, which means all three cones are lit, as we can see in Figure 5. In reality, two of them would have been in shadow, though. Let's look at another example illustrating how to put limits on the effect of directional light sources:

Listing 6

```
#VRML V2.0 utf8
#Example: Using two scoped directional lights

DEF myCone Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.7 0.7 0.1
    }
  }
  geometry Cone { }
}

Transform {
  translation -3 0 0
  children [
    DirectionalLight {
      direction 1 0 0
    }
    USE myCone
  ]
}

Transform {
  translation 3 0 0
  children [
```

```

        DirectionalLight {
            direction -1 0 0
        }
        USE myCone
    ]
}

```

When viewing this example (see Figure 6), note how a `DirectionalLight` is scoped to the group node (in this case a `Transform` node) it is placed in. This feature allows us to make it look like certain objects are in shadow, like the middle cone in our example.

The other kinds of light sources, point lights and spot lights, are a little bit more advanced and also demands more complicated computations for the VRML browser. Of course, they are very useful to increase realism in a many scenes, but they won't be discussed further here. Instead we turn our attention to texture mapping.

2.6 Basic Texture Mapping

When texture mapping is used in a tasteful manner, it often increases the detail and realism in virtual worlds dramatically. In VRML, there are a couple of different nodes supporting texture mapping. We will only look at the `ImageTexture` node, though. This node let us use textures given in the GIF, PNG or the JPEG file format. In its simplest form, it is very easy to use. Look at the following example:

Listing 7

```

#VRML V2.0 utf8
#Example: A Texture Mapped Cylinder

Shape {
    appearance Appearance {
        material Material {
            diffuseColor 0 1 0
            specularColor 0.5 0.5 0.5
            shininess 0.1
        }
        texture ImageTexture {
            url "wood.jpg"
        }
    }
    geometry Cylinder { }
}

```

We specify the texture mapping properties using the `texture` field inside the `Appearance` node. The `url` field is used to give the location of the texture map. Normally, special texture coordinates, often called *s* and *t*, are required at each vertex in a shape if it is going to be texture mapped. In the example above no such coordinates were specified, because all the built in primitives, `Box`, `Cone`, `Cylinder` and `Sphere`, have built in texture coordinates.



Figure 7: A wood texture applied to a cylinder object

Besides the texture information, we also supply a material to the texture mapped shape. Why? In fact, if no material is used, the shape will not be lit and it will lose realism. The material, will blend nicely together with the texture map, for example making highlights possible. The diffuse color of the material, green in our example, is ignored, though. This means we don't have to worry about getting our texture maps messed up with other basic colors.

When we build geometrical models using either the `IndexedFaceSet` node or the `ElevationGrid` node, we can control how the texture will be placed on the models by specifying our own texture coordinates. As an example, let's see how we can put a texture on a triangle polygon:

Listing 8

```
#VRML V2.0 utf8
#Example: How to specify texture coordinates

Shape {
  appearance Appearance {
    material Material {}
    texture ImageTexture {
      url "epcot.jpg"
    }
  }
  geometry IndexedFaceSet {
    coord Coordinate {
```

```

        point [-1 0 0, 1 0 0, 0 1 0 ]
    }
    coordIndex [0, 1, 2, -1]
    texCoord TextureCoordinate {
        point [0 0, 1 0, 0.5 1 ]
    }
}
}

```

As we can see, the texture coordinates are specified in the `texCoord` field of the `IndexFaceSet` node. It is a list of *s* and *t* pair values, hopefully chosen with care. Since there is no `texCoordIndex` field specified in this example, the first texture coordinate belongs to the first vertex, the second texture coordinate belongs to the second vertex, and so on. The resulting world can be seen in Figure 8. Can you see what's on the texture?



Figure 8: A texture mapped triangle polygon

The `TextureCoordinate` node doesn't include anything more than we already used. Anyway, it is defined like this:

```

TextureCoordinate {
    exposedField MFVec2f point []
}

```

Another useful node in this context is the `TextureTransform` node. It is used to apply transformations, like translation, scaling and rotation, to the texture coordinates. For example, this is useful if we want to change how the texture map is placed upon the built in primitives with predetermined texture coordinates. Here is an example using this ability:

Listing 9

```
#VRML V2.0 utf8
#Example: Using the TextureTransform node

Transform {
  translation -2 0 0
  children Shape {
    appearance Appearance {
      material Material {}
      texture ImageTexture {
        url "epcot.jpg"
      }
    }
    geometry Box { }
  }
}

Transform {
  translation 2 0 0
  children Shape {
    appearance Appearance {
      material Material {}
      texture ImageTexture {
        url "epcot.jpg"
      }
      textureTransform TextureTransform {
        scale 2 2
      }
    }
    geometry Box { }
  }
}
```

In this world there are two primitive boxes (see Figure 9). The first one simply uses the built in texture coordinates, resulting in one copy of the texture on every face of the box. However, the built in texture coordinates in the second box are transformed by the `TextureTransform` node added to its appearance description. Since the texture coordinates are scaled in both directions by a factor of two, the texture map appear four times on every side of the box. The `TextureTransform` node is defined as follows:

```
TextureTransform {
  exposedField SFVec2f center 0 0
  exposedField SFFloat rotation 0
  exposedField SFVec2f scale 1 1
  exposedField SFVec2f translation 0 0
```

}

These fields allow us to translate, rotate and scale textures on objects. The `center` field holds the center position about which rotation occurs. If the value is `0 0`, which is default, the rotation point is the lower left corner. To rotate the texture around the textures own center point we have to set the `center` field to `0.5 0.5`.



Figure 9: Transformed texture coordinates

Finally, let's look at the definition of the `ImageTexture` node used in our texture mapping examples:

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

As stated before, the `url` field specifies the location of the image to be used. It might be located locally or even somewhere on the internet. The fields `repeats` and `repeatT` specifies how texture coordinates bigger than one will be treated. We can choose to keep the default settings, which means we let the texture repeat itself in both the *s* and *t* direction, like we saw in the example above, or we can enforce clamping by changing these fields to `FALSE`.

2.7 Controlling the Viewer

So far, we have been navigating around in our virtual world by using the VRML browser, but from what view point do we really want to start? By default, the viewpoint starts at the coordinates 0 0 10, but by using the `Viewpoint` node we can set up, not only where to start, but also define a list of particularly interesting viewing spots. Here is an example:

Listing 10

```
#VRML V2.0 utf8
#Example: Choosing a viewer

DirectionalLight {
  direction 0 -1 0
}

Viewpoint {
  orientation 1 0 0 -0.78
  position 0 5 5
  description "first"
}

Viewpoint {
  fieldOfView 0.6
  orientation 1 1 0 -0.78
  position -4 4 5
  description "second"
}

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1 1 1
    }
  }
  geometry Box { }
}
```

In this scene, there are two different viewpoints. The first one specified will be the starting viewpoint (see Figure 10). In *Cosmo Player* all viewpoints holding a name in the `description` field will be added to a viewpoint list from which the user easily can choose a particular viewpoint. Here is the definition of the `Viewpoint` node:

```
Viewpoint {
  eventIn          SFBool      set_bind
  exposedField     SFFloat     fieldOfView      0.785398
  exposedField     SFBool      jump                   TRUE
  exposedField     SFRotation   orientation         0 0 1 0
  exposedField     SFVec3f      position              0 0 10
  field            SFString     description            ""
```

```

    eventOut      SFTIME      bindTime
    eventOut      SFBool      isBound
}

```

The two fields `position` and `orientation` are used together to specify the view position and direction and the `fieldOfView` field determines the width of the view, in a similar way like a lens in camera does. It is interesting to play around a little with these parameters to get a better feel of their usage.

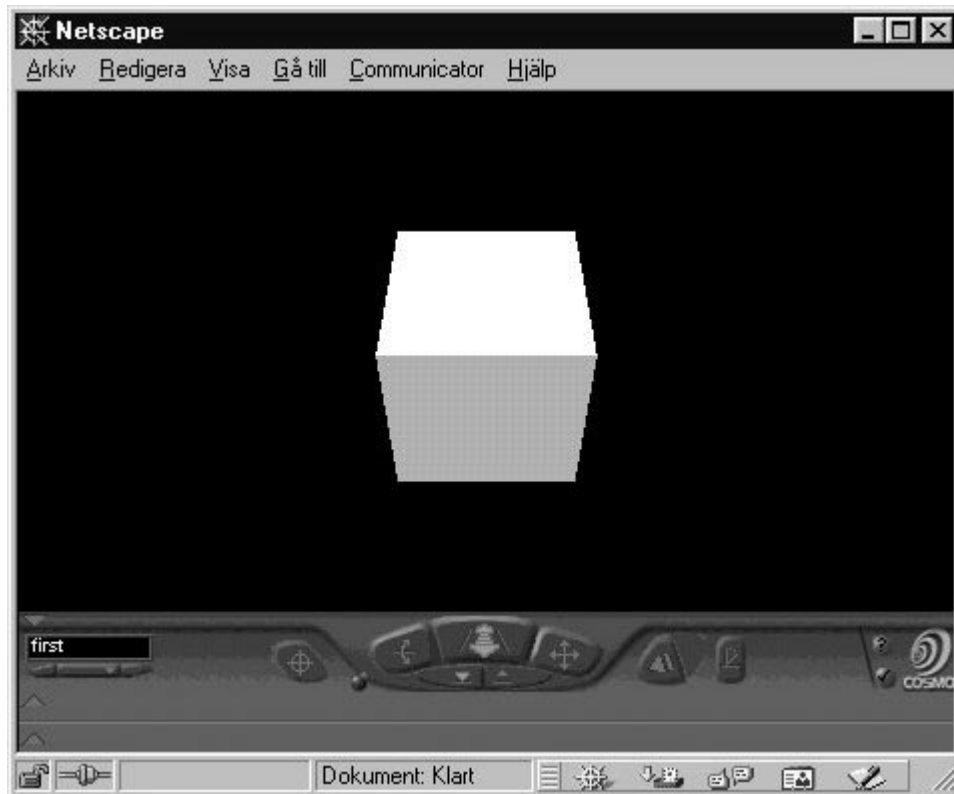


Figure 10: A specified starting viewpoint

This short look at viewpoints concludes our basic discussion of how to start making VRML worlds. Let's move on to look at some interesting optimization possibilities.

3. Optimization Techniques

One very important goal when creating nice looking virtual worlds is to achieve an acceptable frame rate when the world is explored. Although the performance on affordable computers are constantly increasing we still need to be very careful when designing complex virtual environments. By definition, in virtual reality applications, we always need real time performance. The performance achieved depends pretty much upon the way we model our objects as well as the scene. Keeping this in mind, let's take a look at two quite simple possibilities to increase the performance in VRML worlds.

3.1 Adaptive Level of Detail

The first trick is to represent a complex geometrical model in more than one way, from very simple and efficient versions to more and more detailed and realistic versions. Then during rendering a particular representation is chosen depending of the models distance from the viewer. The more far a way an object is, the less detail is needed. There is a direct support for this adaptive level of detail concept in VRML by the node called LOD. It is defined as follows:

```
LOD {
  exposedField      MFNode      level      []
  field             SFVec3f     center      0 0 0
  field             MFFloat     range       []
}
```

The field `level` refers to a list of nodes describing the different representations of an object, the `center` field holds the center position used for all different representation and the `range` field is a list of distance values defining when the browser is going to switch representation. An example will make this clearer:

Listing 11

```
#VRML V2.0 utf8
#Example: Using level of details

DEF my_app Appearance {
  material Material {
    diffuseColor 1 1 0
  }
}

LOD {
  level [
    Shape {
      appearance USE my_app
      geometry Sphere { }
    },
    Shape {
      appearance USE my_app
      geometry Cylinder { }
    },
    Shape {
      appearance USE my_app
      geometry Cone { }
    }
  ]
  center 0 0 0
  range [15, 20]
}
```

In this case we use the LOD node to switch between a sphere, cylinder and cone. Exactly when the switches between these representations happens are specified in the `range` field. The first representation is chosen when the distance between the viewer and the object is less than 15 units (see Figure 11). When the distance is between 15 and 20 units, the second representation is chosen and if the distance is greater than 20 units the last representation is used.



Figure 11: Taking a close look at the LOD node

3.2 Billboards

A well-known technique to decrease the polygon count in a model is to replace a detailed geometrical description by texture mapped polygons. The texture map can give us quite a good experience of some detailed structure. Using this technique, together with the `Billboard` node in VRML, will make it possible to include a lot of objects in a scene that would be far to complex to represent by pure geometrical descriptions. A classical example of this is how to represent trees in a garden. The `Billboard` node groups a list of children that will be forced to face the viewer. Thus, we can represent a tree by a simple flat polygon painted with a tree texture. Let's look at an example world with one billboard polygon and a box object beside it to see how this node works:

Listing 12

```
#VRML V2.0 utf8
#Example: Using billboards
```

```

Billboard {
  children Shape {
    appearance Appearance {
      material Material {}
    }
    geometry IndexedFaceSet {
      coord Coordinate {
        point [-2 -3 0, 2 -3 0, 2 3 0, -2, 3, 0]
      }
      coordIndex [0, 1, 2, 3, -1]
    }
  }
}

Transform {
  translation -5, 0, 0
  children Shape {
    appearance Appearance {
      material Material {}
    }
    geometry Box { }
  }
}

```

Now first take a look at Figure 12. Note that the polygon placed under the Billboard node in the scene graph is always facing towards the viewer, no matter how we navigate around in the VRML browser. (See if you can put a nice tree texture on this polygon by yourself to make this world a little bit more interesting.) The Billboard node is defined like this:

```

Billboard {
  eventIn          MFNode  addChildren
  eventIn          MFNode  removeChildren
  exposedField    SFVec3f  axisOfRotation  0 1 0
  exposedField    MFNode   children        []
  field           SFVec3f  bboxCenter      0 0 0
  field           SFVec3f  bboxSize        -1 -1 -1
}

```

This node is yet another one used to group other nodes together. The most interesting and new part of this node is the ability to define an `axisOfRotation`. The underlying geometry will then be rotated about this axis automatically to get aligned with the viewer.

Of course there are many other things we can do to optimize our worlds. Maybe you come up with some fruitful ideas yourself? For example, what about combining the LOD and the Billboard node? When will such a strategy be useful?

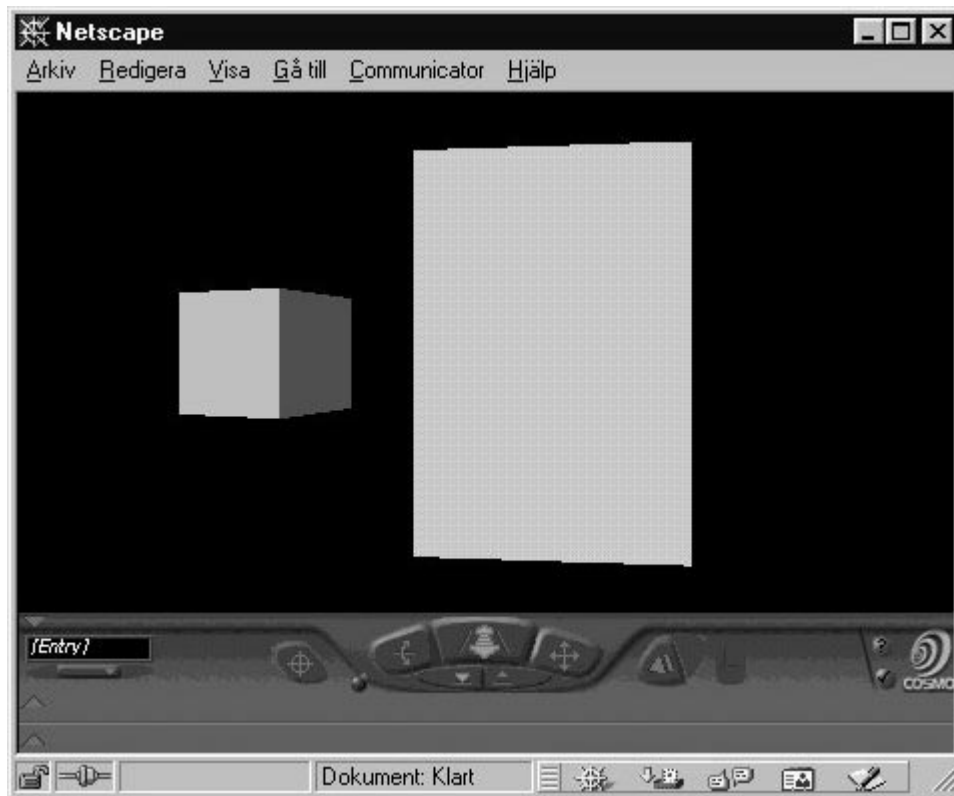


Figure 12: A box and a billboard polygon

4. Moving Beyond Static Worlds

So far, the nature of our world has been static. To make more useful and interesting worlds we need to bring them to life. Some interesting questions are: In what ways can the user interact with the world? How can we create moving objects? In the following sections, just a glimpse of the possibilities are presented.

4.1 Allowing User Interaction

In VRML there are several sensor nodes. These nodes are used to sense some kind of stimuli and translate it into an appropriate VRML event that will affect the scene. For example, the `TouchSensor` node is used to determine when the user touches an object. When that happens, the node generate an event, that will be passed to some receiver node who can react in respond to that event. But exactly who is the receiver? Actually, we have to give that information in our scene description by creating routes, using the `ROUTE` keyword. Hopefully, an example will clarify this:

Listing 13

```
#VRML V2.0 utf8
#Example: Allowing user input

Transform {
```

```

rotation 1 0 0 0.78
children [
  DEF myTouchSensor TouchSensor {},
  DEF myDirectionalLight DirectionalLight {
    direction 1 0 0
    on FALSE
  },
  Shape {
    appearance Appearance {
      material Material { }
    }
    geometry Cylinder { }
  }
]
}

ROUTE myTouchSensor.isOver TO myDirectionalLight.set_on

```

In this scene, there is a directional light source, casting light in the direction of the positive x-axis, but from the beginning it is turned off. By the help of the functionality provided by the TouchSensor node, placed together with the cylinder object, we are able to turn the light on, using the mouse input device (see Figure 13). The ROUTE created at the end of the example specifies that behavior. When the TouchSensor goes off, because of the mouse cursor is placed over the geometry grouped together with it, an event will notify the DirectionalLight node. Note that there has to be a perfect type match between associated events. An eventOut has to have the same type as the corresponding eventIn in the ROUTEs we create. Here is the definition of the TouchSensor:

```

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

```

The eventOut field isOver has the type SFBool. This means we have to connect it to an eventIn field with the same type. Let's look again in the definition of the DirectionalLight node, given in section 2.5, to see if we can find such an eventIn field. Actually, there is no field called set_on defined at all in that node definition. So how does it work then? In fact, there is both an eventIn and an eventOut field implicitly defined for every exposedField in a node. Just add the prefix set_ to find the corresponding eventIn name and the suffix _changed to find the corresponding eventOut name.



Figure 13: Light rays from the left, when the object is in mouse focus

4.2 Introducing Scripts

If we want to add advanced behavior to our worlds, not built into VRML directly, we hopefully have the possibility to accomplish it using a programming language. Even if the VRML97 standard doesn't require that a VRML browser has to support any programming language, we can for example use Java, JavaScript or VRMLScript (a subset of JavaScript) in *Cosmo Player*. We will only use VRMLScript, since it is the easiest of them all and it doesn't have to be pre-compiled. The glue between the scene graph and the code we write is given to us by the `Script` node. Here is an example:

Listing 14

```
#VRML V2.0 utf8
#Example: Using a VRML Script

Transform {
  rotation 1 0 0 0.78
  children [
    DEF myTouchSensor TouchSensor {},
    Shape {
      appearance Appearance {
        material DEF myMaterial Material { }
      }
      geometry Cylinder { }
    }
  ]
}
```

```

    ]
}

DEF myScript Script {
    eventIn SFBool touched
    eventOut SFColor newColor
    field SFBool toggleColor TRUE
    url "vrmlscript:
        function touched() {
            if (toggleColor) {
                newColor = new SFColor(1, 0, 0);
                toggleColor = FALSE;
            } else {
                newColor = new SFColor(0, 1, 0);
                toggleColor = TRUE;
            }
        }
    }"
}

ROUTE myTouchSensor.isOver TO myScript.touched
ROUTE myScript.newColor TO myMaterial.set_diffuseColor

```

Thanks to the script in this scene, the color of the cylinder can be changed dynamically. From the beginning, the cylinder has its default diffuse color. When the mouse cursor enter the area covered by the cylinder its color changes to red and when the mouse cursor leaves this area its color is changed to green (see Figure 14).

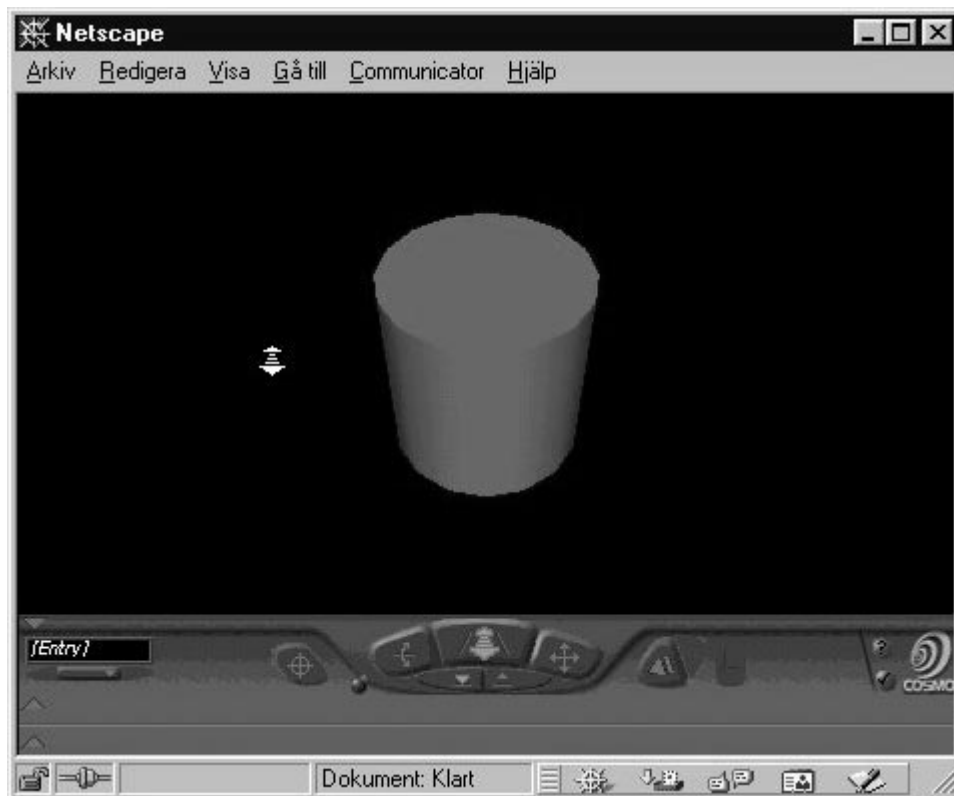


Figure 14: When the mouse cursor left the object it became green

When is this script executed? The code in the script is invoked when the `Script` node receives an `isOver` event from the `TouchSensor` in the scene. Why? Because there is a `ROUTE` statement wiring them together in exactly that way. When this happens, the script function, with the same name as the received `eventIn`, is executed. Furthermore, when an assignment occurs to the `newColor` field in the script code it causes an other event, that will be sent to the `Material` node, causing the change of color. Why? Because there is another `ROUTE` statement connecting them.

5. VRML on the Internet

Remember, the purpose of VRML is to make 3D graphics a natural part of the World Wide Web, as a complement to the widely used HTML standard. We might be able to enrich the on-line experience significantly, if we use virtual reality worlds in a tasteful manner.

One of the most important things on Web pages is the possibility to easily follow hyperlinks. In virtual worlds hyperlinks might be equally important. For this reason, let's take a look at how to insert hyperlinks into our virtual worlds.

5.1 Adding Hyperlinks

Hyperlinks are very easy to place in a scene. To do so, we just use the node called `Anchor`, which let us specify geometry the user can click on in the scene graph. Here is an example:

Listing 15

```
#VRML V2.0 utf8
#Example of how to create a hyperlink

Anchor {
  url "Simple.wrl"
  description "Go to Simple.wrl"
  children Shape {
    appearance Appearance {
      material Material { }
    }
    geometry Sphere { }
  }
}
```

In the `url` field we specify where the hyperlink goes. In this case, it is to the first example world in this tutorial, but we could for example have referred to an image file or a HTML file instead. The `description` field makes it possible for the browser to show what kind of source to expect by following the link (see Figure 15).

The definition of the `Anchor` node reveals it is yet another grouping node, holding a list of children:

```

Anchor {
  eventIn      MFNode      addChilden
  eventIn      MFNode      removeChildren
  exposedField MFNode      children      []
  exposedField SFString    description   ""
  exposedField MFString    parameter     []
  exposedField MFString    url           []
  field        SFVec3f     bboxCenter   0 0 0
  field        SFVec3f     bboxSize      -1 -1 -1
}

```

All the nodes grouped under the `Anchor` node can be clicked on to invoke the hyperlink. The `parameter` field, can be used to pass information to the browser.



Figure 15: The sphere is used as a hyperlink

5.2 Combining VRML and HTML on Web Pages

So far, we have seen our virtual worlds filling the whole page in the Web browser, because they have been directly loaded. Nevertheless, it is also possible to embed 3D worlds into a HTML page using the `<EMBED>` tag in the HTML standard. In this way, a VRML world only needs to occupy a rectangle of a page in the browser. Here is an example:

```
<EMBED SRC="simple.wrl" HEIGHT=400 WIDTH=500>
```

Frames in the HTML standard give us yet another possibility to create Web pages including both VRML worlds and HTML documents. For example, we can have some frames dedicated to view the 3D graphics worlds while other frames are used to give associated 2D graphics information. It is even possible to communicate between such frames, to make things even more interesting.

6. Conclusions

In this tutorial we have started to learn VRML. We have briefly covered many of the basic features and if we put the pieces together we can accomplish quite nice looking worlds. However, it is quite a challenge to master this language. For example, we have only touched upon *sensors*, which are used for triggering events to dynamically affect the scene. We have completely ignored *interpolators*, which are a very useful set of nodes for bringing objects to life. The scripting facilities, all very central to make our virtual worlds more unpredictable and realistic, have been almost completely left out. Hopefully, these and others interesting topics will be discussed in another follow-up tutorial. In any case, now let's have some fun building nice virtual worlds, all sharable over the Internet.

7. Acknowledgements

The author would like to thank Kjell Post for his beneficial suggestions to improve this document.

References

- [Vrm97] The VRML97 specification, www.vrml.org/Specifications/VRML97/
- [Nad97] Introduction to VRML97, Siggraph Tutorial, David Nadeau, John Moreland, Michael Heck, 1997
- [Mar97] Teach Yourself VRML 2 in 21 days, Chris Marrin, Bruce Campbell, Sams.net, 1997