

Studying Geometry, Color and Texture in VRML

Arzu Cöltekin
Jussi Heikkinen
Petri Rönnholm

Arzu.Coltekin@hut.fi
Jussi.Heikkinen@hut.fi
Petri.Ronnholm@hut.fi

Institute of Photogrammetry and Remote Sensing
Helsinki University of Technology

1 INTRODUCTION

In the late '80s at CERN (Center for European Particle Physics) the new hypertext system for the Internet called World Wide Web was developed. Before that the information on the Internet was particularly difficult to access. The invention of hypertext system created links between related data sources, making information easier to access. The hypertext format got a name HTML (Hypertext Markup Language). Quite soon after that raised new idea of creating a virtual reality on the web. And finally in 1994 VRML (Virtual Reality Modeling Language) was announced in the World Wide Web conference in Geneva.

The idea of navigating in 3D virtual reality means that the user uses provided controls to change the viewing conditions. In desktop 3D programs that means calculating the 3D projection of the scene on 2D display surface. This might work as well in the WWW-environment. Although the data rate to be sent through the network would be extremely big. Especially when using the network through the telephone lines would take time and change the meaning of acronym WWW to "World Wide Wait". That is why instead of sending a projection of the scene we could send an instruction to create the 3D-scene through the network and let your computer and VRML-browser do the actual work.

The VRML-format is originally based on the format of *Open Inventor* by Silicon Graphics. It is object oriented. It has a lot of conversion programs and it is a simple ASCII file format. VRML has the most common aspects of *Open Inventor*. Namely:

- Basic Shapes
- Basic material properties
- Basic transformations
- Basic camera views
- Basic texture-mapping
- Basic lightning

In this article we will concentrate on dealing with 3D modeling aspects: Geometry, color, shading and texture of object surface.

1.1 VRML-FILE FORMAT

As mentioned the VRML-code is an ASCII type code. It is highly object-oriented, meaning entities called objects includes properties connected with the object, events and transformations changing the state of the object. Also VRML is a hierarchical language.

A VRML file consists of the following major functional components: the header, the scene graph, the prototypes and event routing. A VRML file always starts with the header. It can be in a following form:

```
#VRML V2.0 <encoding type> [optional comment] <line terminator>
```

Character # denotes usually a comment line and all characters after that will be ignored by the VRML-browser. The “encoding type” for example “utf8” indicates how the internal characters in your VRML models are interpreted.

After the header your VRML-model have to be presented as a “scene graph”, which contains **nodes**. Nodes describe objects and their properties. They are hierarchically grouped and they consist of visual presentation of an object as well as audio-properties which object might have. Also nodes keep inside the event generation mechanism.

Prototypes allow the set of VRML node types to be extended by the user. Prototype definitions can be included in the file in which they are used or defined externally. With prototypes you can define new node types in terms of already defined “built-in” node types or prototypes. What is essential is that all node type names should be unique in each VRML file. In a prototype interface declaration by using a PROTO statement you can define the event interface as well as default values for prototype’s fields.

Each prototype instance can be considered to be a complete copy of the prototype, with its own fields, events and copy of the prototype definition. A prototyped node type is instantiated using standard node syntax. For example, as a following way:

```
PROTO Cube [ ] { Box { } }
```

In this example prototype has an empty interface declaration and it can be instantiated in scene graph as follows:

```
Shape { geometry Cube { } }
```

With the event routing mechanism you can build the connection between node generating an event and node receiving an event. So it is an interface of a model to interact with a user. Event routing gives you a mechanism to separate event from the scene graph hierarchy. Once generated, events are sent to their routed destinations in time order and processed by the receiving node. This processing can change the state of the node, generate additional events, or change the structure of the scene graph. The standard event routing has following structure:

```
ROUTE <name>.<field/eventName> TO <name>.<field/eventName>
```

Script nodes allow author-defined event processing. An event received by a Script node causes the execution of a function within a script. This is a way to create the dynamically changing VRML-model. Within a script you can dynamically direct events for any node existing in your model. The Script node can be defined as a following way:

```
Script {  
    url [ "http://foo.com/myScript.js",  
        "javascript: function foo() { ... }" ]  
}
```

In this article we will concentrate on basic built-in construction models in VRML rather than event driven interface of the model.

2 GEOMETRY

VRML defines the unit of measure of the world coordinate system to be in meters. The angles will be measured in radians instead of degrees, time in seconds and colors in RGB-values with values between 0.0 and 1.0. In VRML-models a Cartesian, right-handed, three-dimensional coordinate system will be used. By default, the viewer is on the Z-axis looking toward the origin in opposite direction of Z-axis. The positive direction of the X-axis will be to the right and positive Y-axis is to straight up.

In the node structure Shape nodes include two parts, namely geometry and appearance node. In this section we will concentrate on *geometry* nodes, *appearance* node will be discussed in following sections.

```
Shape {  
    appearance NULL  
    geometry NULL  
}
```

The Shape node and the Background node are the only two nodes, which can be rendered i.e. projected on screen respect to viewing conditions.

2.1 GEOMETRY NODES

The VRML 2.0 defines the following primitive shapes: **Box**, **Cone**, **Cylinder**, and **Sphere**. More advanced shapes are: **Elevation Grid**, **Extrusion**, **IndexedFaceSet**, **IndexedLineSet** and **PointSet**. The values or properties of several geometry nodes can contain property nodes such as *Coordinate*, *Color*, *Normal* and *TextureCoordinate*. The geometric property nodes are defined as individual nodes so that sharing is possible between different geometry nodes inside Scene graph.

The *Coordinate* node presents a set of 3D point coordinate values. The single coordinate values are separated by space and points are separated by commas. The *Coordinate* node is used in **PointSet**, **IndexedLineSet** and **IndexedFaceSet** geometry nodes.

```
Coordinate { point [ 0 0 0, 1 1 1 ] }
```

The structure of the *Color* node is similar. Single color values (0.0 - 1.0) are separated by space and indexed color sets are separated by commas. This node appears inside **PointSet**, **IndexedLineSet**, **IndexedFaceSet** and **ElevationGrid**. The *Normal* node appears in **IndexedFaceSet** and **ElevationGrid** and it presents the surface unit normal vectors in corresponding surface point. The syntax of the *Normal* node follows the previous formula.

```
Color { color [ ] }  
Normal { vector [ ] }
```

The *TextureCoordinate* node instead presenting a property in a single surface point presents the correspondence between texture patch coordinates, *IndexedFaceSet* and *ElevationGrid* points. More detailed presentation of texture mapping will be shown in the following sections.

2.2 PRIMITIVE SHAPES

The whole idea with VRML is to provide the geometric data as compact form as possible. In these sense parametric primitives as: cubes, cylinders, cones and spheres are essential for that purpose. They provide all information to construct the object within few parameters.

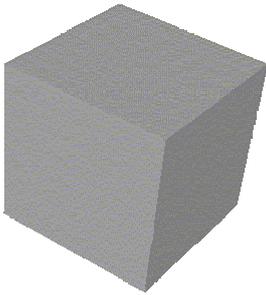


Image 2.1 Box { size 2.0 2.0 2.0 }

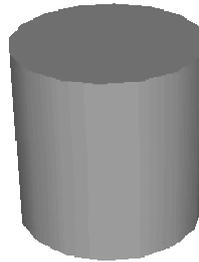


Image 2.3 Cylinder { radius 1.0
height 2.0
side TRUE
bottom TRUE
top TRUE
}

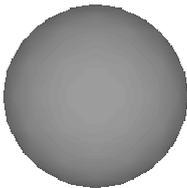


Image 2.2 Sphere { radius 1.0 }

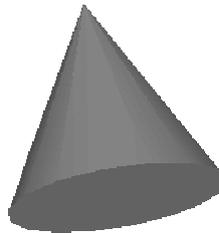


Image 2.4 Cone { bottomRadius 1.0
height 2.0
side TRUE
bottom TRUE }

The additional parameters *side* and *bottom* with *Cone* and *Cylinder* nodes define if the created object is a solid model or just an object having an indefinitely thin surface. If both parameters are set TRUE, which is also default value, the object is considered as closed object. In an opposite case object is considered to be open depending the parameter which is set as FALSE.

2.3 ADVANCED STRUCTURES

More complicated surfaces need different type of data structures. Parametric surface structures, as splines would be appropriate to present uneven variation in an object shape. The NURBS (nonuniform rational B-splines) would store surface parameters in very compact way. The reason that these structures are not included in VRML, even they are widely used in CAD-based designing, is that VRML-models are supposed to be viewed with real-time speed. Rendering a NURBS based structures need a lot more computing than wireframe-type surface models.

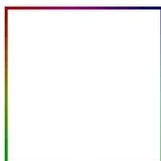
2.3.1 POINTSET

Perhaps the simplest structure to apply for uneven spaced points is a **PointSet**, which only can present an unevenly spaced 3D point cloud. The structure does not provide any tools to present any topological information. Points can be visually separated from each other by color indexing.

```
Shape{ appearance Appearance { material Material {emissiveColor 1 1 1 }}
  geometry PointSet {
    coord Coordinate {point [
      1.0 1.0 1.0
      -1.0 -1.0 -1.0
      1.0 -1.0 1.0
    ]
  }
  color Color { color [
    1.0 1.0 1.0
    1.0 1.0 1.0
    1.0 1.0 1.0
  ]
}
}
```

2.3.2 INDEXEDLINESET

By using **IndexedLineSet** node structures you can create 3D polylines. The structure includes topology information in separate field so the point coordinates have to be defined only once. The point coordinates will be defined in a local coordinate system.



```
Shape{
  appearance Appearance {
    material Material {emissiveColor 1 1 1 }}
  geometry IndexedLineSet {
    coord Coordinate {
      point [ 0 0 0, 1 0 0, 1 1 0, 0 1 0 ] }
    coordIndex [0 1 2 3 0]
    color Color { color [1 0 0, 0 1 0, 0 0 1] }
    colorIndex [0 1 2 0 1]
    colorPerVertex TRUE
  }
}
```

Image 2.5 IndexedLineSet

The *Coordinate* field specifies the point coordinates on polyline and *coordIndex* field indicated in which order lines are connected. With such a structure you are able to create set of non-connecting line sets. To separate the sets of indexes from two adjacent polylines the marker -1 is used, i.e. an index of -1 indicates that the current polyline has ended and the next one begins.

The field *colorPerVertex* is a Boolean value, which indicates if the color specified in *color* field will be applied to each vertex. If the value is set TRUE and the line is a polyline, in a case that succeeding vertexes have be assigned different colors, the color will be applied as a gradually changing from vertex to an other. In an opposite situation, color will be assigned to whole polyline indicated by the first index value in *colorIndex* field.

2.3.3 INDEXEDFACESET

The *IndexedFaceSet* node has a similar type of structure as *IndexedLineSet*. The *Coordinate* field specifies a set of 3D points. The planar faces in the local coordinate system are then specified with in a *coordIndex* field. The list of coordinate indexes defines the faces to be drawn. To separate the indexes from a face the marker -1 is used, i.e. an index of -1 indicates that the current face has ended and the next one begins. Because the faces are always defined by closed polylines, you don't need to define the first point twice.

```
Shape{
  appearance Appearance { material Material { }}
  geometry IndexedFaceSet {
    coord Coordinate { point [-1 -1 -1, 1 -1 -1,
                             1 1 -1, -1 1 -1,
                             -1 -1 1, 1 -1 1,
                             1 1 1, -1 1 1] }
    coordIndex [3 2 1 0 -1, 4 5 6 7 -1,
              0 1 5 4 -1, 5 1 2 6 -1,
              7 6 2 3 -1, 0 4 7 3 ]
    color Color { color [1 0 0, 0 1 0, 0 0 1]}
    colorIndex [0 2 1 0 1 0]
    colorPerVertex FALSE
    creaseAngle 0
    solid TRUE
    ccw TRUE
    convex TRUE
  }
}
```

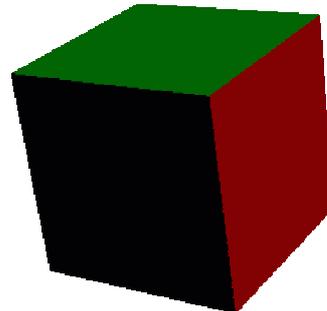


Image 2.6 IndexedFaceSet

The color indexing follows the same approach as *IndexedLineSet* node with an exception that indexes denotes the face sets instead of lines. The *creaseAngle* denotes how the edges between adjacent faces are handled. If the angle between these two faces is smaller than given threshold, the edge will be visible. If the angle is wider than the junction between the adjacent faces, the edge will be drawn smoothly by the browser.

The Boolean flag *ccw* indicates if the coordinates in *coordIndex* field are given in counterclockwise order and the *solid* flag tells if the both sides of the face should be visible or not.

The *Convex* field denotes if the faces defined in the *coordIndex* field are convex or not. VRML can only handle convex faces. With concave faces the browser has to create small faces inside the defined face and this will hinder the processing.

2.3.4 ELEVATIONGRID

The *ElevationGrid* node can be used in a similar way as the previous *IndexedFaceSet* node. With the *ElevationGrid* node equally spaced data points are used instead of a coordinate list. This kind of approach decreases amount of the data to be transmitted.

```
Shape { appearance Appearance {material Material {
}}
  geometry ElevationGrid{
    xDimension 4
    zDimension 4
    xSpacing 1
    zSpacing 1
    height [ 0 0 0 0, 0 1 1 0,
            0 1 1 0, 0 0 0 0]
    ccw TRUE
    solid TRUE
    creaseAngle 0 }}

```

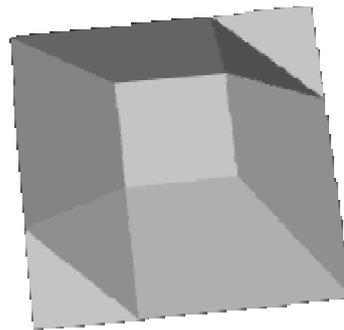


Image 2.7 ElevationGrid

The grid is determined on XZ-plane and *xDimension* and *zDimension* fields determine the number of points included in the grid. The *xSpacing* and *zSpacing* fields specify the real extent of the grid by determining the equal spaces between adjacent surface points. The *height* field presents the height values in each grid node from left to right and from top to bottom.

As can be seen from the example of *ElevationGrid*, the symmetric object has not been constructed, as it should. VRML defines the form of the object and the browser has to construct and render it based on the code. Unfortunately the browser used here has not succeeded in this task as can be seen in the picture.

Both *IndexedFaceSet* and *ElevationGrid* node support also *TextureCoordinate* and *Normal* node structures. The *TextureCoordinate* node handles how the texture elements are mapped on the surface. *Normal* node indicates the surface normal direction with unit normal vectors in surface points or in elevation grid nodes.

2.3.5 EXTRUSION

The *IndexedFaceSet* is a fine tool for modeling complicated surfaces. The problem with such a structure is that you need to define a lot of points to present the surface precisely. The fascinating approach is to apply defined 2D-plane section and move it around the surface of the object. The nominal shape of the plane patch and route how the plane shape has been applied on the surface will be stored in this structure.

The *crossSection* field defines the 2D-shape with 2D vectors forming a cross section of the shape. The cross section will be determined on the XZ-plane. In a case of cube the cross section shape is a square.

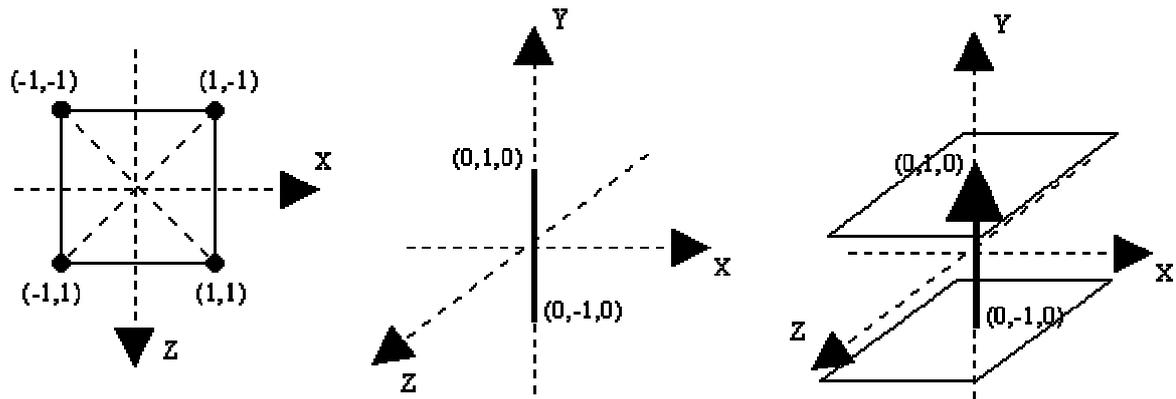


Image 2.8 Moving a cross section along the spine. [Fernandes, 1998]

The *spine* defines the path that the cross section will travel to create the shape and in *orientation* field you can determine the orientation of the shape in different parts of the path. Otherwise it will follow the tangent of the spine path. By default the orientation of the shape coincide with Y-axis.

```
Shape { appearance Appearance {material Material {
}}
  geometry Extrusion{
    crossSection [-1 -1, -1 1, 1 1, 1 -1, -1 -1]
    spine [0 -1 0, 0 1 0]
    orientation [0 0 1 0, 0 0 1 0]
    scale [1 1, .5 .5]
    beginCap FALSE
    endCap FALSE
    ccw TRUE
    convex TRUE
    solid FALSE
    creaseAngle 0
  }
}
```

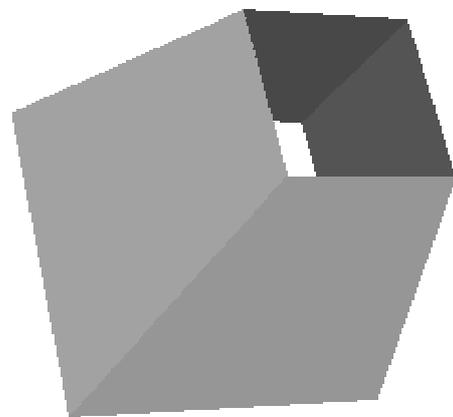


Image 2.9 Extrusion

With *scale* values you can change the size of the shape in different parts of the spine. Boolean flags *beginCap* and *endCap* determine whether the shape on beginning and end of the spine will be visible. Also a *solid* field conjunction with *beginCap* and *endCap* will effect if the object constructed is a solid or a tube kind of object.

The **Extrusion** type of object construction is similar kind of method as “sweeping”, which is widely used in CAD-based modeler.

2.4 COMPLICATED MODELS

Until now we have only talk about single objects in their local coordinate system. How shall we tie up the objects to present the whole scene? The *grouping* is the key word. The *group* node lets you treat a set of nodes as a single entity. It is a kind of encapsulation, where you can tie up all those nodes together which you would like to have same properties: same material node or same transformation.

```

Group {
  children [
    Shape {
      appearance Appearance {material Material {}}
      geometry Cylinder {
        height 5.0
        radius 0.5
      }
    }
    Shape {
      appearance Appearance {material Material {}}
      geometry Sphere {}
    }
  ]
}

```

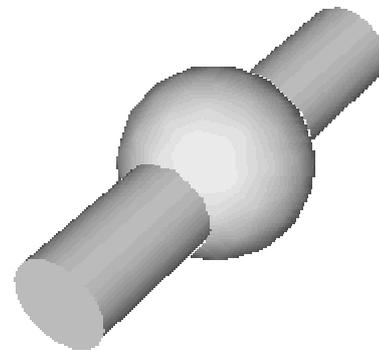


Image 2.10 Grouped objects

The **Group** node field *children* contains all the nodes included in the group. The *bboxCenter* specifies the center of a box that encloses the nodes in the group. The value for this field is a 3D point. The *bboxSize* specifies the size of a box that encloses the nodes in the group. By default this field has a value of -1 -1 -1, which implies that no box is defined. The values for this field must be greater than or equal to zero. If the children nodes do not fit inside the box defined the results are undefined. But the values of these two fields are optional and the browser can understand the structure without them. They only provide a tool for optimizing the performance of the browser.

The **Transform** node is a group node. This node allows you to define a new local coordinate system for the nodes within the group. You can apply *Scale*, *Rotation* and *Translation* transformation to all the nodes inside a **Transform** group. The **Transform** group defines a transformation between a local coordinate system and an outer coordinate system.

The *scale* specifies a 3D scaling transformation and *scaleOrientation* field defines a rotation of the axes. The *center* field defines the center of the scaling transform. The *rotation* field defines a rotation on an arbitrary axis. A vector and an angle define a rotation uniquely. The 3D vector specifies the axis of rotation, where the angle specifies the amount to rotate in a counterclockwise direction. The *translation* field defines the origin of the local coordinate

system respect to the outer coordinate system. The syntax of the *Transform* node is following:

```
Transform {  
    scale 1 1 1  
    scaleOrientation 0 0 1 0  
    center 0 0 0  
    rotation 0 0 1 0  
    translation 0 0 0  
    bboxCenter 0 0 0  
    bboxSize -1 -1 -1  
    children []  
}
```

The result of transformation of a 3D object depends on which order transformations are made. In VRML the order is *Scale* => *Rotation* => *Translation*.

VRML also allows you to define a set of nodes, or a node with particular field values, as a new node type. This is called instancing a node. This has clear advantages when, for instance, you would like to repeat the object in your scene multiple times. Defining the node once and using it multiple times you have to change only the node, which is being defined instead of changing all occurrences of the defined node. The mechanism for instancing uses DEF and USE structure.

```
Shape { appearance  
    DEF common_appearance Appearance {  
        material Material {diffuseColor 1 0 0} }  
    geometry Sphere {} }  
Transform {  
    translation -2 0 0  
    children [  
        Shape {  
            appearance USE common_appearance  
            geometry Cone {}  
        } ] }
```

In many cases the VRML models are actually build in some specialized 3D modeling software and then converted to VRML-code. The converters transform the model into VRML in predetermined way. Hardly any of those modelers can though add any interaction to the model, even though it is supported by VRML. Also geometrical conversion can apply in different ways by using different structures to present the model.

The presented ideas in this section have been based on the following references: [Fernandes, 1998; Carey et al. 1997; Fox, Shaddock, 1996].

3 COLORS AND MATERIALS

The VRML uses the RGB color model (Red, Green, and Blue). All values are between 0.0 and 1.0. For instance (0 0 0) is black, (0 0 1) is blue, and (1 1 1) is white (Image 3.1). For a single geometric shape there is a **Color** node, where multiple colors can be defined for example for an indexed set of faces (**IndexedFaceSet**).

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

For example, if six colors are defined, the **Color** node looks like this:

```
color Color { # Six colors: (blue, green, cyan, red, magenta and yellow)  
  color [ 0 0 1, 0 1 0, 0 1 1, 1 0 0, 1 0 1, 1 1 0 ]  
}
```

Now if an **IndexedFaceSet** of 12 faces is defined, these colors can be added to every face with the color index:

```
colorIndex [ 0, 1, 1, 0, 2, 3, 3, 2, 4, 5, 5, 4 ]
```

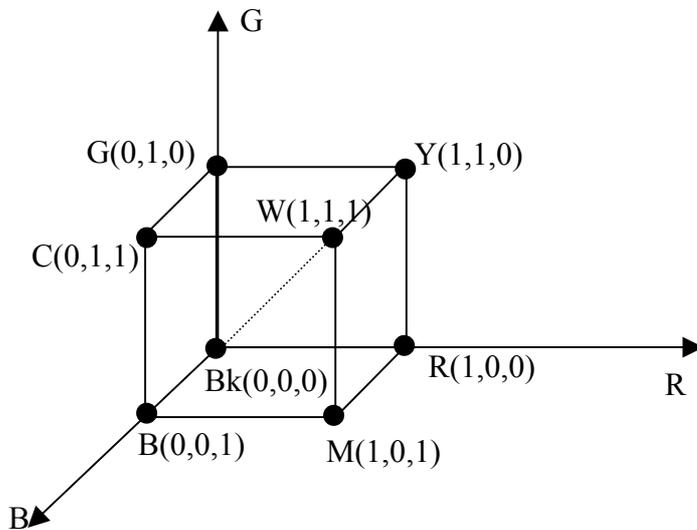


Image 3.1 The RGB color model, where R = red, G = green, B = blue, M = magenta, C = cyan, Y = yellow, Bk = black and W = white. [Hearn, Baker, 1986]

If only **Color** node is used to present an object, the result is not necessarily very realistic one. In VRML there is no possibility to chose material like “steel”, “rubber”, “wood” etc. Instead of those only colors, light reflections and transparency are used to simulate all the materials. For this VRML has a **Material** node that has six fields: *diffuseColor*, *emissiveColor*, *ambientIntensity*, *shininess*, *specularColor*, and *transparency*. All of them affect also to the color presentation.

The ***ambientIntensity***: This field specifies the amount of ambient light reflected by the object. There is more information about ambient light in section 4.1. Ambient color is calculated as ***ambientIntensity*** × ***diffuseColor***.

The ***diffuseColor***: This field defines the color of the object. The idea is that when the light ray strikes the surface, some parts of the light's color spectrum are absorbed. For example a blue surface absorbs all but the blue part of the color spectrum. This field is however ignored when using colored textures. The ***diffuseColor*** field reflects all VRML light sources. It depends on the angle of the surface with respect to the light source. The more directly the surface faces the light, the more diffuse light reflects. In section 4 there is more about lightning.

The ***specularColor***: In this field the color of the shiny spots of the object can be defined. A brightly shining spot makes an object to look like a polished one. The ***specularColor*** is added when the angle from the light to the surface is close enough to the angle from the surface to the viewer.

The ***shininess***: Shininess is often associated with smooth surfaces. When shininess value is low, glow on the object is soft and covers bigger area. Bigger shininess values cause sharper and smaller highlights.

The ***emissiveColor***: With this field it is possible to model glowing objects. This possibility can be useful for example for displaying pre-lit models when the light energy of the room needs to be computed explicitly or for displaying scientific data [Carey et al.,1997].

The ***transparency***: This field controls the transparency of the object. It has a fractional value between 0 and 1. A value of 0.0 sets the related object completely opaque and a value of 1.0 makes the object completely transparent what means that the object is totally invisible. Transparency is calculated as a percentage mixture between the color of the object and the color of the background behind it.

Syntax in the ***Material*** node is:

```
Material {  
    diffuseColor 0.8 0.8 0.8          # [0,1]  
    ambientIntensity 0.2             # [0,1]  
    emissiveColor 0.0 0.0 0.0        # [0,1]  
    specularColor 0.0 0.0 0.0        # [0,1]  
    shininess 0.2                    # [0,1]  
    transparency 0.0                 # [0,1]  
}
```

All the "Color" fields in the ***Material*** node have an RGB value associated that must be between 0.0 and 1.0.

4 LIGHTING

Lighting determines how much light strikes an object and partially how much should be reflected. Also the *Material* node, that was introduced in previous section, affects to reflections. In the VRML there are four different light sources: ambient light, spot lights, directional lights and point lights. Furthermore, it is possible to switch the lighting off. Then all objects are painted with RGB colors without any modulation by the lighting.

A precise description of the VRML's lighting equations can be found from International Standard ISO/IEC 14772-1:1997 [Carey et al., 1997] in section 4.14.

4.1 AMBIENT LIGHT

An ambient light coats all objects in a scene with the same level of light. In the real world there are billions of light rays bounce randomly from one surface to another. That is still too much for computers to calculate quickly. Therefore light ray bouncing is simulated (and greatly simplified) with ambient light. The VRML allows defining ambient light separately for every object.

The ambient light prevents areas of scene that are in shadow from rendering completely black. If ambient light level is too high, all the objects will be evenly lighted and they look flat and one-dimensional.

4.2 POINT LIGHT

Point lights radiate equally in all directions like the sun, lamps and matchsticks (Image 4.1). Light rays travel in straight lines until they meet a surface. In the VRML 2.0 there is also possibility to define attenuation for light rays and radius within the objects are illuminated. A 3D location of a point light source is defined in the local coordinate system. Point lights are defined in the *PointLight* node.

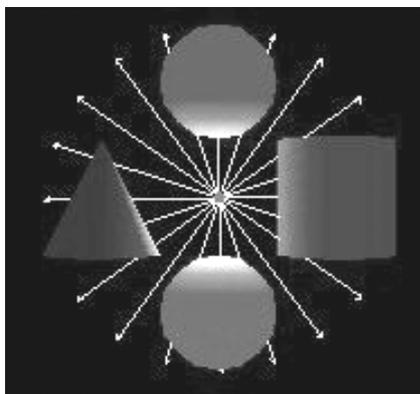


Image 4.1 Point Light. [Fernandes, 1998]

Syntax for point light is:

```
PointLight {  
    on TRUE  
    intensity 1          # [0,1]  
    ambientIntensity 0  # [0,1]  
    color 1 1 1         # [0,1]  
    location 0 0 0      # [-∞, ∞]  
    attenuation 1 0 0  # [0, ∞]  
    radius 100          # [0, ∞]  
}
```

4.3 SPOT LIGHT

The **SpotLight** node defines a light source that emits light from a specific point and is pointed at a particular direction within a solid angle (Image 4.2, A). The cone of light is defined by two fields: the **cutOffAngle** that defines the angle of the cone in radians; and the **beamWidth** that defines the angle of an inner cone within the light intensity is full and constant (Image 4.2, B). Between the inner cone and the outer cone the intensity of the light rays decrease. Precise equations for this can be found from International Standard ISO/IEC 14772-1:1997 [Carey et al.,1997] in section 6.45. If the **beamWidth** is larger than the **cutOffAngle** then the light has a constant intensity within the cone. The **radius** node defines the maximum distance from location that may be illuminated by the light source.

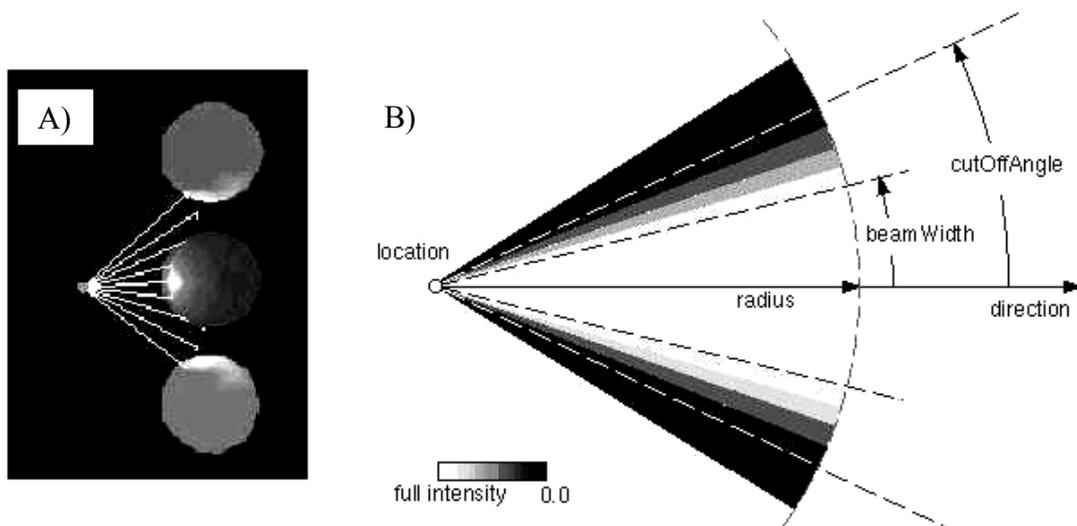


Image 4.2 A) Spot light and B) the cone of the light. [Fernandes, 1998; Carey et al.,1997]

Syntax for spot light is:

```
SpotLight {  
    on TRUE  
    intensity 1 # [0,1]  
    ambientIntensity 0 # [0,1]  
    color 1 1 1 # [0,1]  
    location 0 0 0 # [-∞, ∞]  
    direction 0 0 0 # [-∞, ∞]  
    attenuation 1 0 0 # [0, ∞]  
    radius 100 # [0, ∞]  
    cutOffAngle 0.78 # [0, π/2]  
    beamWidth 1.57 # [0, π/2]  
}
```

4.4 DIRECTIONAL LIGHT

The *DirectionalLight* node defines a directional light. The light rays are emitted along parallel rays from an infinite distance away (Image 4.3). Unlike for point lights or spot lights there are no attenuation or radius available for directional light. This light illuminates only the nodes, which are defined within the same group. Objects placed outside the group are not lit. [Fernandes, 1998; Carey et al.,1997]

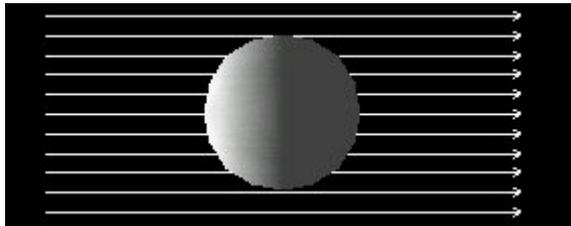


Image 4.3 Directional light. [Fernandes, 1998]

Syntax for directional light is:

```
DirectionalLight {  
    ambientIntensity 0 # [0,1]  
    color 1 1 1 # [0,1]  
    direction 0 0 -1 # [-∞, ∞]  
    intensity 1 # [0,1]  
    on TRUE  
}
```

4.5 SCANLINE RENDERING

The VRML browsers use scanline rendering. In scanline rendering the path of the light ray is traced from the light source to the object and back. So, there are no interactions between objects. That is why the VRML can not truly reproduce reflections, shadows or refraction. [Fox, Shaddock, 1996]

5 SHADING

Shading polygonal models provide increasing levels of realism. Shading determines how the colors are spread across the surface. Three main methods are flat shading, Gouraud shading and Phong shading. Phong-shaded models are the most photorealistic ones. Unfortunately calculation time increases when more detail are added. This is one reason why VRML worlds still look quite unrealistic. The type of shading is depended on the browser but if an object is going to be shaded by having lights shine on it, it must have a *Material* node defined as part of its appearance.

5.1 FLAT SHADING

Flat shading is the easiest way to assign color to the faces of polygons. However, because the entire face is given a single color, objects look entirely flat and so the result looks unrealistic. In flat shading, all the points on one face are oriented in the same direction. This direction of the face is called surface normal (Image 5.1). If light sources affect so that all differently oriented polygonal surfaces are shaded differently, the result is more realistic. On polygons that lie on flat surfaces, flat shading is very useable. Unfortunately there are rarely perfectly flat surfaces in nature and there can be even millions of surface normals in one surface.

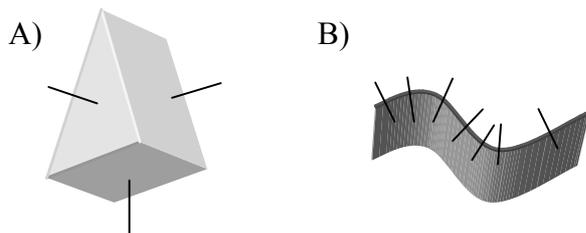


Image 5.1 A) Surface normals on flat faces. B) Some surface normals on a curved surface.

5.2 GOURAUD SHADING

With the Gouraud shading it is possible to present more realistic curved surfaces compared to the flat shading. In the Gouraud shading intensity discontinuities between adjacent planes of a surface representation are removed. The idea, how to compute the normal at one vertex, is to average the normals at all the patches sharing this particular vertex (Image 5.2). The color of the faces is interpolated between the vertexes so the colors blend smoothly together. A more detailed description about this color interpolation can be found from book Computer Graphics by Hearn and Baker [Hearn, Baker, 1986] pages 289-291. The results are quite nice looking and computational time is relatively small. However, if compared to the flat shading, the Gouraud shading is more time consuming.

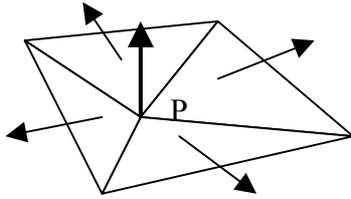


Image 5.2 In the Gouraud shading normal vector at the vertex point **P** is calculated as the average of the surface normals for each plane. [Hearn, Baker, 1986]

The Gouraud shading has also some disadvantages. Especially highlights on the surface are sometimes distorted. Furthermore the linear intensity interpolation can cause bright or dark intensity streaks on the surfaces. These unpleasant effects can be reduced by using a greater number of polygon faces or by using completely other method. [Hearn, Baker, 1986]

5.3 PHONG SHADING

The Phong shading provides some improvements to the Gouraud shading. In Phong shading the colors are not interpolated but the normals. At each point on the scan line (Image 5.3) is the interpolated normal and the final color for a point is computed from it. This method produces great results but is very slow. [Hearn, Baker, 1986]

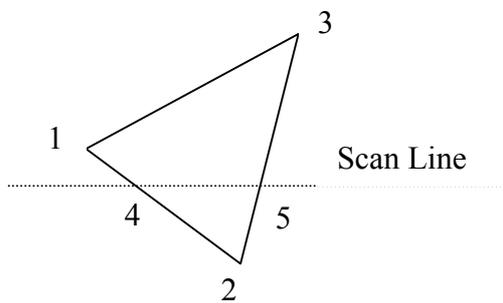


Image 5.3 A scan line is used for interpolating shading. [Hearn, Baker, 1986]

6 TEXTURE MAPPING IN VRML 2.0

Texture mapping in VRML is basically the same as texture mapping in all other areas of 3D graphics. It is all based on the same fundamental concepts.

The Texture Nodes in VRML 2.0 are:

ImageTexture: defines a still texture map using an image file.

MovieTexture: defines a moving texture map using a movie file.

PixelTexture: defines a still texture map made from explicit pixel values.

TextureTransform: defines a 2D transformation applied to texture coordinates.

Appearance: where the texture nodes live.

Shape: where the Appearance node lives.

TextureCoordinate: defines a set of 2D coordinates to be used to map textures to the vertices of subsequent geometry nodes like IndexedFaceSet or ElevationGrid.

In the VRML 2.0 format, the Texture node exists as part of an Appearance node. Material, texture, and texture transform are always related to one another. Also, the Appearance node exists inside of a Shape node. This associates a specific appearance with a specific geometrical object (in the example below, a cube). No other object in the file will have this appearance unless specified by the programmer.

This example maps the given image file (somefile.jpg) onto a cube.

```
#VRML V2.0 utf8
Group {
  children  Shape {
    appearance Appearance {
      texture ImageTexture {
        url  [ "somefile.jpg" ]
      }
    }
    geometry Box {}
  }
}
```

6.1 SIMPLE SHAPES

Here a sphere, a cone, a cylinder and a box may be seen with applied image textures.

The sphere looks like the map has been "shrink wrapped" around it. The cone and the cylinder stretch the map around their vertical sides and place cut-outs of the map on their flat ends. The cube gets a full copy of the map on each of its six faces.



Image 6.1 This is texture mapping at the most basic level.

6.2 TEXTURES APPLIED ALONG A MATERIAL

Texture images may be one component (greyscale), two component (greyscale plus transparency), three component (full RGB color), or four-component (full RGB color plus transparency). An ideal VRML implementation will use the texture image to modify the diffuse color and transparency of an object's material (specified in a Material node), then perform any lighting calculations using the rest of the object's material properties with the modified diffuse color to produce the final image. The texture image modifies the diffuse color and transparency depending on how many components are in the image, as follows:

1. Diffuse color is multiplied by the greyscale values in the texture image.
2. Diffuse color is multiplied by the greyscale values in the texture image; material transparency is multiplied by transparency values in texture image.
3. RGB colors in the texture image replace the material's diffuse color.
4. RGB colors in the texture image replace the material's diffuse color; transparency values in the texture image replace the material's transparency.

6.3 THE APPEARANCE NODE

The Appearance node is called from the appearance field of a Shape node. This is where most of the nodes that effect the way an object looks are kept. It's not really a texture node, but since it's where our texture nodes live, we should discuss it briefly. Here is a small sample:

```

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

```

As you can see, it has three fields; material, texture, and textureTransform. These fields take a SFNode as a value. This means that instead of a numerical value, these fields take another node as a value. Also, these are exposedFields so they can be changed from a script.

The material field is the home of the Material node. This node allows you to set color, transparency, shininess, etc. Our concern however, is with the other two fields. The syntax is a little confusing at first because the field names are very similar to the names of the nodes that go in them. For instance, to use a TextureTransform node, the syntax would look something like this...

```

Appearance {
  ...
  textureTransform TextureTransform {
    ...
  }
}

```

The texture field is where ImageTexture, MovieTexture, and PixelTexture nodes go. You can use any of these nodes here, but you can only use one texture node per object. If you need part of an object textured with an image and another part textured with a movie file or another image, you'll have to break up your object into separate parts and map them individually.

The textureTransform field takes the TextureTransform node as it's value. You can only have one set of transforms per texture map.

6.4 THE THREE TEXTURE NODES

In the VRML 2.0 - the ImageTexture node maps the still images and the PixelTexture node maps arrays of pixels. In addition to images and pixels, VRML 2.0 has a third node for mapping movie files onto objects - the MovieTexture node.

6.5 COMMON GROUND

There are two basic fields that the Texture nodes have in common: the URL field and the Repeat field.

6.6 THE IMAGETEXTURE NODE

The ImageTexture node defines a texture map by specifying an image file and general parameters for mapping to geometry. Texture maps are defined in a 2D coordinate system, (s,

t), that ranges from 0.0 to 1.0 in both directions. The bottom edge of the image corresponds to the S-axis of the texture map, and left edge of the image corresponds to the T-axis of the texture map. The lower-left pixel of the image corresponds to $s=0, t=0$, and the top-right pixel of the image corresponds to $s=1, t=1$.

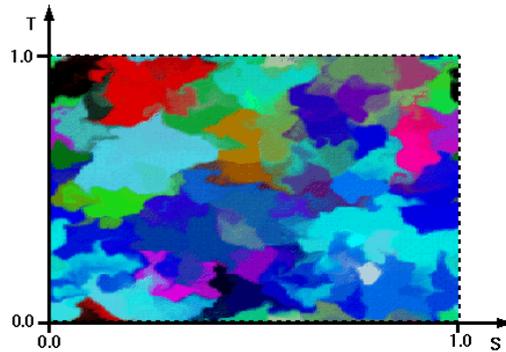


Image 6.2

The syntax is simple:

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

ImageTexture is made up completely of the two fields url and repeat. So the most basic use of ImageTexture would look something like this...

```
ImageTexture {
    url "someimage.jpg"
}
```

6.7 THE PIXELTEXTURE NODE

The PixelTexture node defines a 2D image-based texture map as an explicit array of pixel values and parameters controlling tiling repetition of the texture onto geometry.

Texture maps are defined in a 2D coordinate system, (s, t), that ranges from 0.0 to 1.0 in both directions. The bottom edge of the pixel image corresponds to the S-axis of the texture map, and left edge of the pixel image corresponds to the T-axis of the texture map. The lower-left pixel of the pixel image corresponds to $s=0, t=0$, and the top-right pixel of the image corresponds to $s=1, t=1$.

It is same as ImageTexture node, the only difference is that you define the image in your code in a matrix form instead reading from a file.

```
PixelTexture {
```

```

    exposedField SFImage image 0 0 0
    field SFBool repeatS TRUE
    field SFBool repeatT TRUE
}

```

Sometimes you may see PixelTextures referred to as "inline textures".

6.8 THE MOVIE TEXTURE NODE

The MovieTexture node extends the texture mapping capabilities of VRML 2.0 by giving us the ability to make them move.

```

MovieTexture {
    exposedField SFBool loop FALSE
    exposedField SFFloat speed 1
    exposedField SFTime startTime 0
    exposedField SFTime stopTime 0
    exposedField MFString url []
    field SFBool repeatS TRUE
    field SFBool repeatT TRUE
    eventOut SFFloat duration_changed
    eventOut SFBool isActive
}

```

The key field here is the url which is where the name of the movie file goes.

6.9 THE TEXTURE TRANSFORM NODE

The TextureTransform node gives us the ability to position the texture map on the object with a great deal of precision. Here is the syntax:

```

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

```

In the interest of accuracy, the TextureTransform node performs its transformations on the texture coordinates rather than on the texture itself.

The four fields that give us our texture transformations are center, rotation, scale, and transform.

6.10 USING THE TEXTURECOORDINATE NODE TO MAP COMPLEX OBJECTS

The VRML 2.0 browser should map polygonal objects using the object's bounding box (an imaginary box that completely contains the object within it). What we need is a way to explicitly define how the texture map will be placed on the object. Placing textures on complex polygonal objects is what the TextureCoordinate node is all about. Node looks like this:

```
TextureCoordinate {  
  exposedField MFVec2f point []  
}
```

The TextureCoordinate node gives us a way to "hang" a texture on an object by connecting each vertex in the object with a corresponding u,v value on the texture map.

An example can be seen below.

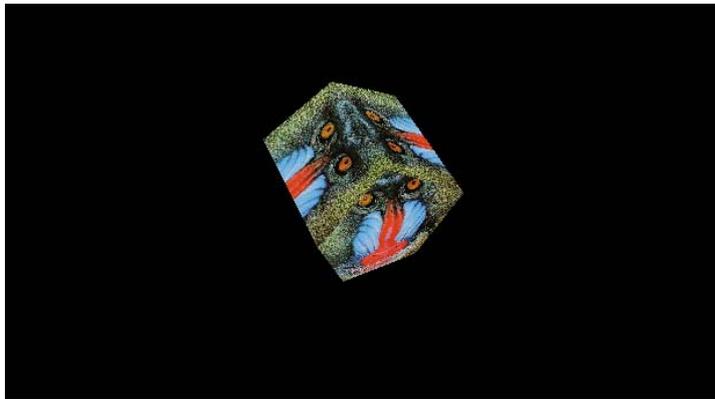


Image 6.3 Textured Complex Shaped

6.11 INSTANCING TEXTURES WITH DEF AND USE

DEF and USE are the two declarations that allow us to instance things. Instancing allows us to define a chunk of VRML once (using DEF) and then reuse it (with USE) throughout the file. Instancing gives us a convenient way to duplicate things throughout the VRML file. It could result as a faster download and some memory savings.

6.12 DISCUSSION

Although, as it is said in the beginning of this chapter, the fundamentals of texture mapping in VRML is not different then in the computer graphics, the mapping science's nature leads us to analyze the accuracy of images used for texture-mapping purposes. In this concept, the question is mostly before 'importing' the images to VRML file to make 'photo-realistic' looks. Current VRML browsers do not have many interactive utilities as in CAD systems -

such as view control, area, volume calculations. Their main function is to visualize, and at that point visual correctness of the images can be questioned.

K. Temfli classifies the steps for Texture Mapping as (1998):

- . Enhance Images
- . Compute Image Polygons
- . Check and Edit
- . Cut and Geometrically Rectify
- . Retouch
- . Homogenize
- . Store
- . Paste (→ VRML)

Several concerns around this steps and possible solutions may be as follows:

The size of the data used for texture mapping. Using more and more images to enrich your virtual world may result unpleasantly and your world may not be usable remotely because of the amount of the data would make the data transfer too slowly.

The effect of the large data sizes on the speed of navigation, can be reduced to some extent by adjusting the views with different resolutions. Sensors can be used for defining certain viewpoints that tells to the browsers to choose the low resolution image since the user is close enough to the object. From a distance the user doesn't likely need to see many details in the pictures. If a copy of the same texture-image is saved in a low resolution, and this is called first, when the viewpoint gets closer to the world, the better resolution of the particular object(s) can be called. This way the whole world would not be so 'heavy'.

The angle of view matters, because when one takes a picture, there is a certain angle that your camera *sees* the target and surrounding. If one uses this picture to texture i.e. a cube-shaped box, when you move around you will notice that things don't look like as they were supposed to be from this view.

The angle of view problem is a known problem for photogrammetrists and other people who use images to make measurements from another aspect. Here we are concerned about visualization, but the answer goes slightly to the same area: correcting the images. In some of the CAD programs (e.g. TargetJr) it is possible to control the view. But within VRML environment we do not have this opportunity to control 'imported texture-images'. The only answer seems to register (also used the term: rectify or correct) the image relatively to the model. This way you transform the texture to the model coordinate system. This gives the opportunity to "glue the wrapping paper" to your surface and walk away. It would still look ok. But this process has to be done within some other environments- or a script should be written to perform this task.

Differing light effects is one of the common problems when you have to make a 'mosaic' of more than one image. If you think of a cube-shaped box, and for instance that you want to 'wrap' it with its real texture, you will have to take several images of the cube. Simply because there's no other way to see all the faces- but you do have all the faces (maybe excluding the bottom) and you'll be able to move around the box. So, that means, you will use different images to extract the piece to texture for each face. When you have your final

box textured, you may notice that it doesn't look like the same box from different faces because the light is different in the image. There is nothing you can do about it in this environment. You can edit a light source somewhere, but it will not help the lights in the picture (image).

The effect of differing lights is also a known problem from the earlier uses of images. The suggestion is to preprocess the images to get to somewhat in same lightning (enhancements such as normalization, equalization etc.) in an "image processing" environment first and then use them as textures.

Storing the textures has also to be considered. The way you store the image files relates the 'speed' of the client's browsing.

Storing the texture data may be also pretty tiresome. One can store those in one file, multiple files or in a database. A database for those images seems to be the best solution.

7 REFERENCES

Carey, R., Garvin, B. and Marrin, C., 1997, ISO/IEC 14772-1:1997, Virtual Reality Modeling Language, (VRML97), <http://www.vrml.org/Specifications/VRML97/>.

Fernandes, A., 1998, VRML Interactive Tutorial, Software, Interaction & Multimedia (SIM), Departamento de Informática, Universidade do Minho, Portugal, <http://tom.di.uminho.pt/vrmltut/toc.html>

Fox, D., Shaddock, P. 1996, Web Publisher's Construction Kit, Waite Group Press, CA, USA

Hearn, D., Baker, M. P., 1986, Computer Graphics, PRENTICE-HALL International, USA, p. 352.

Reed-Ballreich, C. & 3Name3D, 1996, Texture Mapping in VRML 2.0, <http://www.ywd.com>

Smith, J., 1999, Floppy's VRML Guide, <http://www.ee.surrey.ac.uk/Personal/ee41rs/vrmlguide/index.html>

Tempfli, K., 1998, 3D-GIS, Moniulotteinen mallintaminen ja visualisointi , Mittaus- ja kartoitustekniikan tutkijakoulutuskurssi (1. Jakso). Toim Jaakko Santala. Teknillisen korkeakoulun Geodesian ja kartografian laboratorion julkaisuja 33, s. 63-123