3D WEB GRAPHICS WITHOUT PLUGINS USING VML

A Master Project

Presented to

The faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Jiewei Lin

Advisor: Dr. Chris Pollett

August 2003

ABSTRACT

Today, a plugin is required to view 3D graphics in the most common web browsers, Internet Explorer and Netscape. The closest thing that exists is LiveGraphics3D [MK01], a Java 1.1 applet for viewing Mathematica objects and a weak XML language that is translated using Javascript to VML in Internet Explorer [GR00]. VML (Vector Markup Language) [W3C98] is an XML [W3C98] based mark-up language for vector graphics. It is natively supported by Internet Explorer 5 and above, the most commonly used web browsers today. The goal of this project is to develop a stylesheet-transformation from the X3D language [W3DC01] to VML.

This project provides an Internet Explorer solution to plugin-less viewing of 3D graphics. A similar project was developed by [PS02] but with the target language SVG [W3C01] which is a stronger vector mark-up language. It provides a Netscape solution to plugin-less viewing of 3D graphics. X3D is a version of VRML (virtual reality modeling language) [ANM97] specified as an XML DTD. It is a W3C standard and supports a robust set of tags for describing 3D objects and their behaviors. A stylesheet, which transforms from X3D to VML, allows viewing 3D graphics on the web without plugins. This style sheet saves the users time and energy to download and install the plugins.

3D objects that appear on the screen are put together by 2D-polygons, such as triangles. These polygons are typically shaded so that the surface of the object appears smooth and

the underlying polygons are not noticeable. To draw the 3D object on a 2D plane in a particular orientation a projection is applied to the object. The VML language supports drawing of 2D-shapes such as polygons, at the given locations on the screen with a limited support for affine transformations of these shapes. It also supports gradient painting of shapes. These tools can be used to render 3D-objects. Taking an X3D file and producing a VML image of a particular view of the object it represents, involves generating a mesh corresponding to the X3D object, calculating shading and doing the final 2D projection.

This project is written using style-sheet transformations. Basically, these are rules, which are applied to a tag before and after the tag is read. The Internet Explorer supports the stylesheet transformation language XSLT (Extensible Stylesheet Language Transformations) [W3C99]. XSLT supports tag-replacement by other tags as well as manipulating tag-attributes. These transformations are tied to ECMAscript (aka Javascript) code [ECMA99] to manipulate matrices.

Table of Contents

1. **INTRODUCTION**

X3D is a W3C standard for 3D graphics on the web. It requires plug-ins to view in Internet Explorer and Netscape, the most common web browsers. VML is an XML application for vector graphics that can be embedded in HTML pages, where GIF and JPEG images appear. Vector graphics take up less space and so can be downloaded much faster than GIF and JPEG over the network [HR01]. The translator is an XSLT stylesheet to provide an Internet Explorer solution to plugin-less viewing of X3D graphics. It does so by taking a 3D document as an input and transforms it into a VML document which appears like a 3D image.

The core stylesheet uses object-oriented programming. It has thirteen classes. They are classes Shape, Box, Cylinder, Cone, Sphere, Vector, Point, Intensity, RGB, TransformationMatrix, ProjectedPolygon, Group, and Transform. The UML diagram of these classes is in Section 4.2. Detailed descriptions of each class member variable and function are in Appendix A.

The primitive shapes are represented by points. The points are organized into two dimensional arrays. The points at the same level are grouped sequentially in the same array and all points at the beginning of all arrays for each primitive shape start at the same column as shown in the next example. Suppose there is a box. From the box's

width, height, and depth, its eight corners can be found. The box below is formed by points 0 – 7.

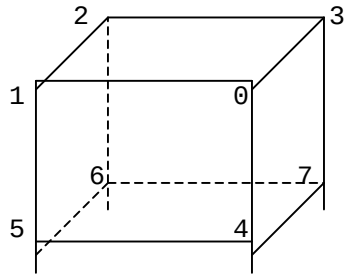


Figure 1-1 A box is represented by eight points.

These eight points are organized into a 2X4 array. The points 0, 1, 2, and 3 form row0, and Points 4, 5, 6, and 7 form row1. So the points of a box represented by a 2X4 array are as follows:

Box array:

	Column[0]	*Column[1]*	*Column[2]*	*Column[3]*
row[0]	P0	P1	P2	P3
row[1]	P4	P5	P6	P7

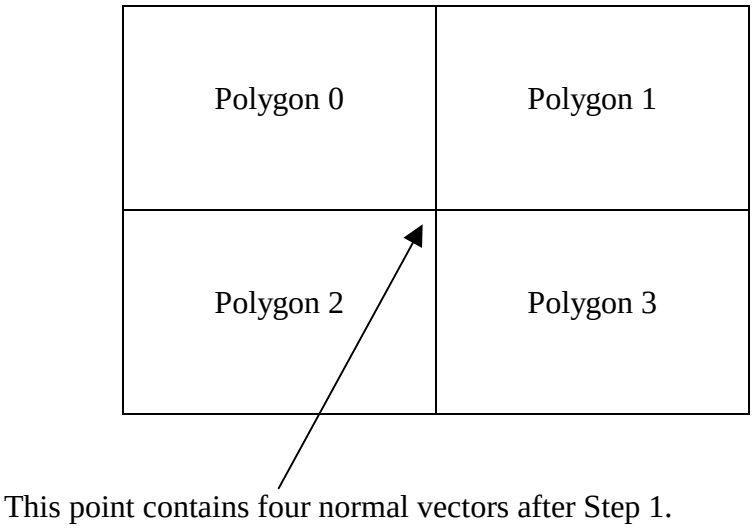
Section 6.1.2 describes how to calculate points for each primitive shape and how to represent them.

The object is translated by specifying a three-dimensional translation vector, which determines how much the object is to be moved in each of the three coordinate directions. Similarly, the object is scaled with the three coordinate scaling factors [HB97]. Rotation is complicated, but can be simplified using a formula which handles a general case. A 4X4 matrix called TransformationMatrix is used to represent transformations. Please refer to Section 6.1.5 for details.

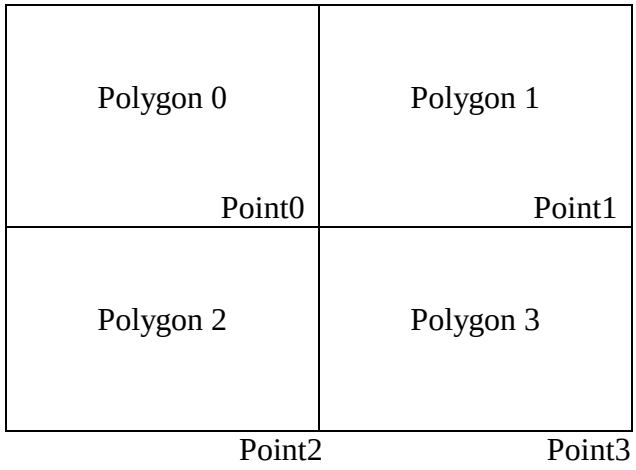
The Phong lighting model is used to calculate intensity for a point on a surface. The Phong lighting model gives a realistic effect. The factors such as the amount of ambient, diffuse, and specular light need to specify for this lighting model. Section 6.1.3 explains the formula to calculate intensity using the Phong lighting model.

Gouraud shading is used to render polygon surfaces. Gouraud shading renders polygons by linearly interpolating intensity values across the surface [HB97]. For each polygon, find the two intensities and the two colors corresponding to the two intensities. This procedure is implemented in five steps.

- First of all, find a normal vector for each polygon and record the vector in fours points of the polygon. Each point in mesh is connected to four polygons (see figure).



- Second, average the normal vectors of the polygons adjacent to a given point.
- Third, average the two normal vectors. In the diagram below, the average of the normal vectors of Point 0 and Point 2 become the first normal vector for Polygon 3. The average of the normal vectors of Point 1 and Point 3 forms second normal vector for Polygon3. Polygon 3 has two normal vectors now. These vectors will be used to calculate shading since VML's gradient tag only shading between two colors.



- Fourth, find two colors according to the two intensities for each polygon.
- Finally, use VML's gradient tag to render polygons.

Section 6.1.4 explains each step in detail.

The translator has grouping features. The grouping tags consist of <Group> and <Transform> tags. They allow the manipulation of a group of shapes together. The transformation matrix is used to represent each <Transform> tag and do nothing for each <Group> tag. Section 6.1.6 will explain how we implement grouping.

The generation of VML tags is challenging. XSLT is not a DOM document so we cannot create VML tags directly inside JavaScript section of X3dToVml.xsl. This was done by concatenating all necessary information into a super long string and parsing it later. This is also a limitation as will be mentioned in Section 8. Section 6.1.8 explains how to generate VML tags.

The translator becomes more interesting after the user interface is added. A new file X3dToVml.html is added as part of the translator, mainly to handle user interface events. The file also replaces USE attributes with their corresponding DEF nodes when the XSLT processor initially loads an X3D document. The DEF and USE attributes are very useful for marking up large scenes where components can be reused. Section 6.2 explains what X3dToVml.html is used for and how it works.

The maximum number of polygons the translator can render is small. It renders up to about 500 polygons. It takes 344 milliseconds to render 72 polygons, four seconds to render 360 polygons, and 96 seconds to render 576 polygons. Section 7 has results of the tests.

The translator successfully transforms an X3D document to a VML document using XSLT. X3D <Group>, <Transform>, <Shape>, <Appearance>, <Box>, <Cylinder>, <Cone>, and <Sphere> tags are supported. The users can also transform scenes using the user interface buttons.

Clipping is left for future work. The translator displays all polygons now. The shapes and polygons are generated and rendered even if they are background objects. The translator might get faster if it clips at the object and polygon level. This is left for future work.

This document is organized as follows: Section 2 introduces three-dimensional graphic concepts. Section 3 gives an introduction to XML, X3D, VML, XSLT, JavaScript and DTD. Section 4 shows the design of the translator. Section 5 lists the requirements to use the translator. Section 6 explains the implemention in detail. Section 7 shows the maximum load. Section 8 lists the limitations. And Section 9 ends the report with a conclusion.

2. THREE-DIMENSIONAL GRAPHIC CONCEPTS

The material for this section follows the book [HB97]. To get a display of a three-dimensional scene in world coordinates, we first need to set up a view reference. This view reference is like a camera. It defines the position and orientation of a three-

dimensional scene to be viewed. The objects are then moved to view reference coordinates and projected onto the selected display plane. Finally, the object is displayed in wire frame or shaded surfaces with color.

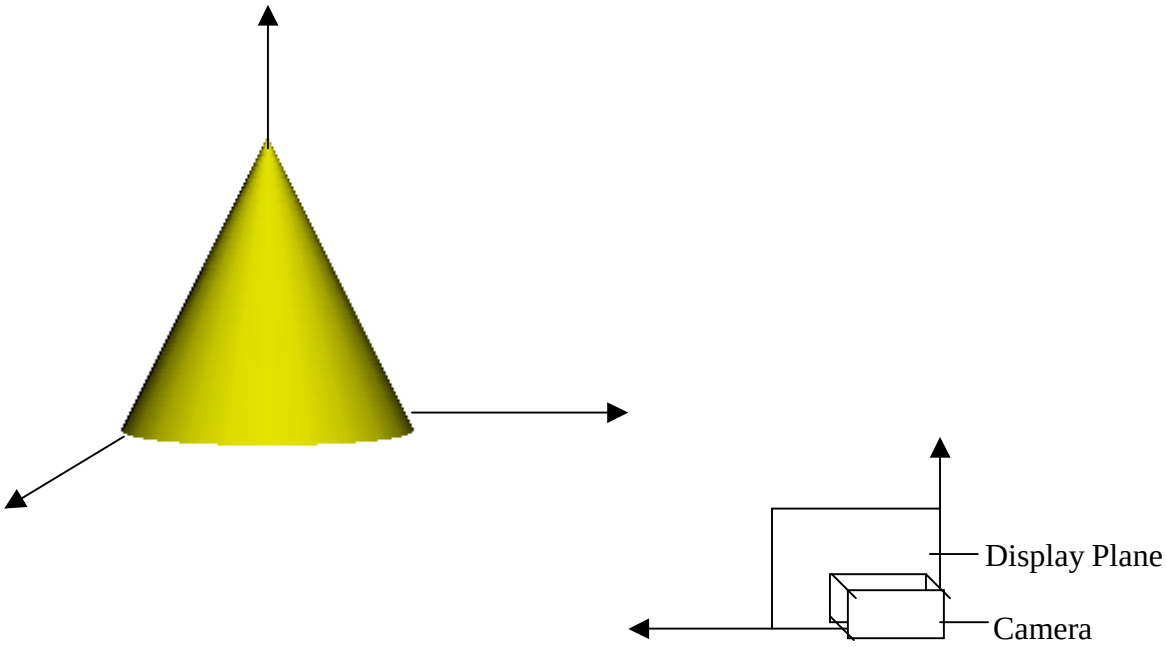


Figure 2-1. Coordinate reference for getting a view of a 3D scene.

Transformations

Three-dimensional rotations are more flexible than two-dimensional rotations. For two-dimensional rotations, the objects can be rotated about an axis that is perpendicular to the xy-plane. In three-dimensional space, the objects can be rotated about any axis. Figure 2-2 is an example of a chair after rotating in three-dimensional space.

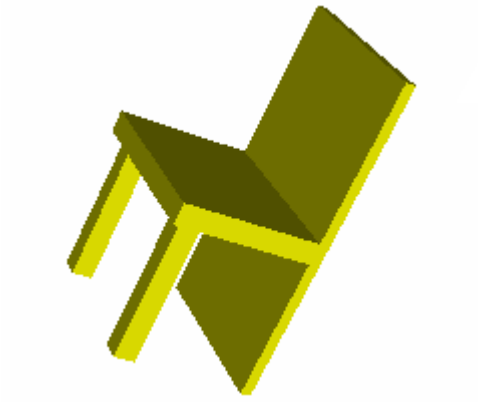


Figure 2-2. A view of a chair rotated in three dimensional space.

Projection

Parallel projection is one way to create a view of a three-dimensional object. Parallel lines are preserved on the display after a parallel projection. An example is shown in Figure 2-3.

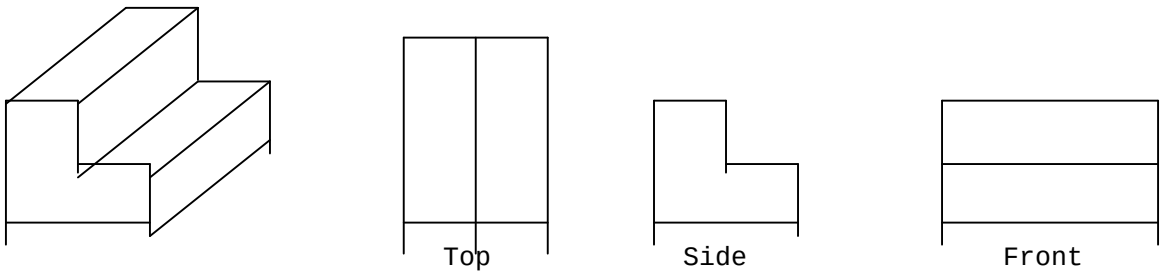


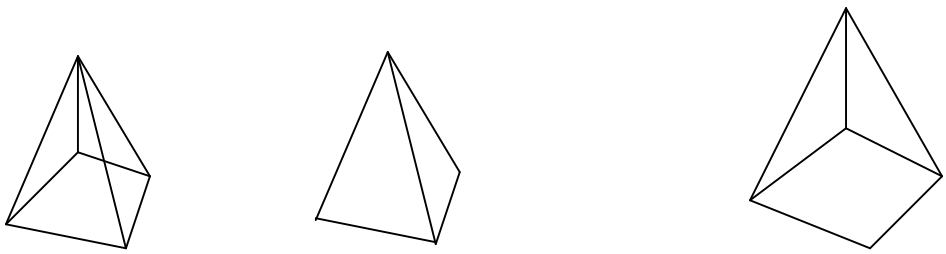
Figure 2-3. Three parallel projection views of an object.

Another way to create a view of a three-dimensional scene is to project objects using perspective projection. Perspective projection projects points to the display plane along converging paths. If the objects are the same size, farther objects appear smaller and closer object appear bigger. In a perspective projection, parallel lines in a scene do not stay parallel on the display plane. Perspective projection produces more realistic views. Figure 2-4 is a perspective view of a column of archways.

Figure 2-4. A perspective view of a column of archways.

Depth Cueing

Depth information allows easy identification of the front and back of an object. An object might appear ambiguious without depth information, as in Figure 2-5a.



a) No depth information. b) Downward view. c) Upward view.

Figure 2-5. Wire frame view of a) containing no depth information, and b), c) containing depth information.

Depth information can be displayed at the object, polygon, line, and point level. Usually a lower level gives more accurate depth information.

Surface Rendering

Objects look more realistic if they are rendered according to lighting and their surface characteristics. The light comes from the intensity, light sources, and background light. Figure 2-6 is a simple surface rendering of a living room.

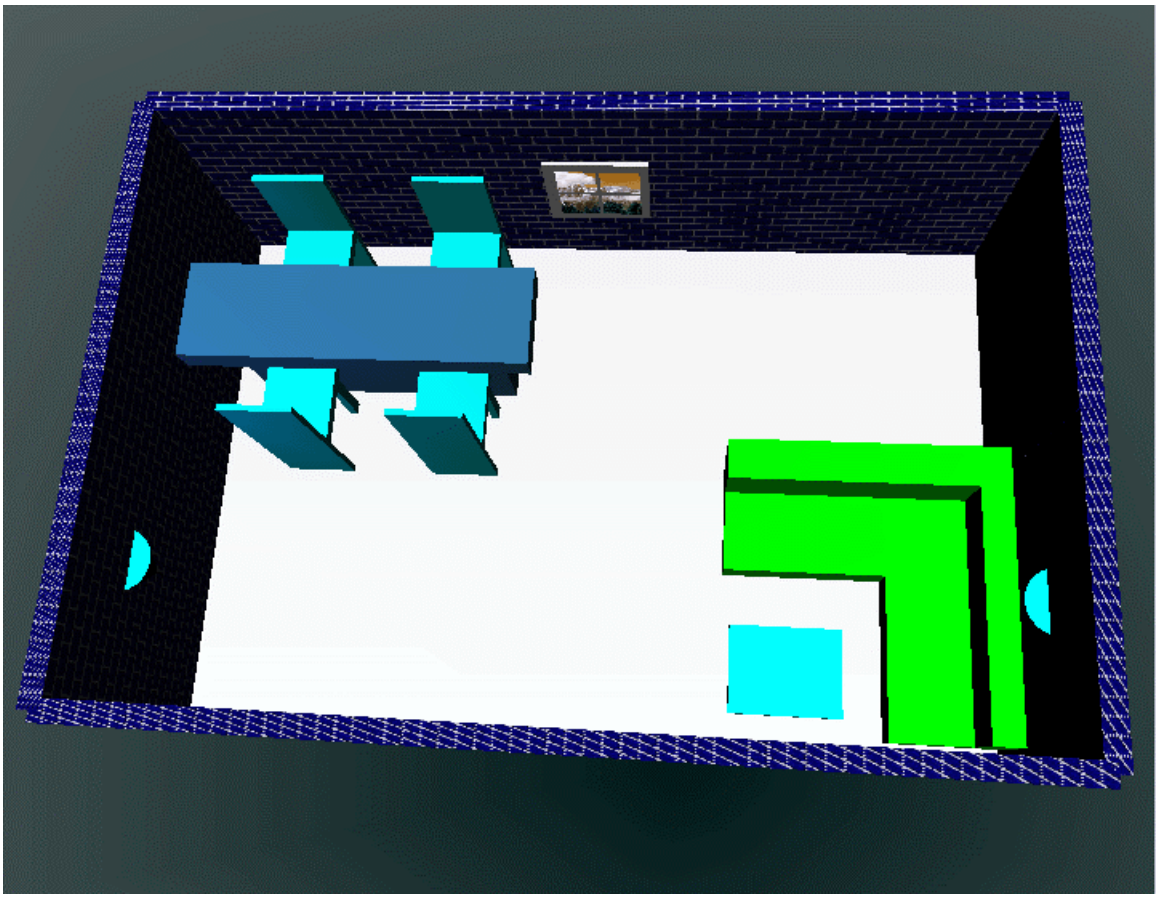


Figure 2-6. A simple surface rendering of a living room.

3. INTRODUCTION TO XML, X3D, VML, XSLT, JavaScript AND DTD

Before we dive into the translator, let's look at the basic concepts of XML, X3D, VML, XSLT, JavaScript and DTD.

3.1 XML

XML (Extensible Markup Language) is a subset of SGML (Standard Generalized Markup Language). XML eliminates much of the complexity of SGML, but has the same goal as SGML. XML can mark up any type of data, for example Scalable Vector Graphics (SVG), MathML, and Chemical Markup Language (CML). XML describes a syntax that can be used to create one's custom languages [HD00].

Rules for elements (page 31 of [HD00]):

- Every start-tag must have a matching end-tag
- Tags cannot overlap
- XML documents can have only one root element
- Element names must obey XML naming conventions
- XML is case-sensitive
- XML will keep white space in your text

The following code meets all the rules for a well-formed XML file.

```
<!-- top element, histograms may contains 0 or more histogram tags
-->
<histograms>
<!-- a histogram may contain 1 or more bins,
     binwidth specifies bin width
     maxNumber specifies the maximum bin height
-->
<histogram binwidth="15" maxNumber="100">
    <!-- number specifics height of each bin -->
    <bin number="100" />
    <bin number="0" />
    <bin number="70" />
    <bin number="88" />
    <bin number="90" />
</histogram>
</histograms>
```

The above code contains a root element <histograms>. The <histograms> element contains a <histogram> element, and the <histogram> element contains five bin elements.

3.2 X3D

X3D is an XML language. Three-dimensional objects in virtual worlds can be built using X3D. A virtual world can be any fragment of the real world one can think of, for example, a tree, a park, or a city. The predefined primitive shapes Box, Cone, Cylinder, and Sphere are used to construct these worlds in X3D. The translator supports all these primitive shapes plus Group and Transform tags. An example of a box that has width of 200.0, height of 300.0, and depth of 400.0 in X3D is given below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="X3d2Vml.xsl"?>
<!DOCTYPE X3D SYSTEM " X3D2VML.dtd">
<X3D>
      <Scene>
            <Shape>
                  <Appearance><Material/></Appearance>
                  <Box size="200.0 300.0 400.0" />
            </Shape>
      </Scene>
</X3D>
```

The first line of the above source code states that this file is a standalone version 1.0 XML document. The second line specifies to use `"X3d2Vml.xsl"` to process the document. The third line indicates that this XML document is to be validated by a DTD file called “X3D2VML.dtd” in the current directory as this file. <X3D> … </X3D> tag are required for all X3D documents. The Appearance tag specifies an object’s color and surface texture. The default is a glowing white appearance. A shape of a box that has width of 200.0, height of 300.0, and depth of 400.0 is the only object in this world.

3.3 VML

VML is an XML application for vector graphics that can be embedded in HTML pages where GIF and JPEG images appear. Vector graphics take up less space and thus can be

downloaded much faster than GIF and JPEG over the network [HR01]. VML is natively supported by Internet Explorer 5.0 and later.

The Polyline and fill tags are the core tags used for this translator. The Polyline tag is a series of straight lines drawn between successive pairs of points. The Fill tag specifies how and with what the shape is filled. Gradient is an attribute value of the Type attribute in element Fill [HR01]. Following is sample code using these tags.

```
<html xmlns:v="urn:schemas-microsoft-com:vml"
xmlns:o="urn:schemas-microsoft-com:office:office"
xmlns="http://www.w3.org/TR/REC-html40">

<head>
<style>
v\:* {behavior:url(#default#VML);}
</style>
</head>

<body>

<v:polyline
    print="false"
   points="181pt,154pt,126pt,140pt,126pt,180pt,181pt,214pt,181pt,154pt"
   fill="true"
   fillcolor="blue">
<v:stroke on="false"/>
<v:fill method="linear sigma" angle="45" type="gradient" />
</v:polyline>

</body>

</html>
```

The html root element above binds the namespace prefix v to the URI `urn:schemas-microsoft-com:vml`. Similarly, The head element above contains a style element indicating that VML's default renderer will be used for display. The VML renderer is a program installed with Internet Explorer 5 and later [HR01]. The Polyline draws a polygon with four points with no outlines. The fill element tells the VML renderer to fill

this shape with gradient color blue at a 45 degree angle. The resulting VML is shown below.

Figure 3-1. Sample VML code using the Polyline and fill tags.

3.4 XSLT

XML documents usually contain data only. They do not include any formatting information. The information in an XML document may not be in the desired form to be presented. XSL provides information on how to present or process the XML document [HD00]. You may view their relations as below:

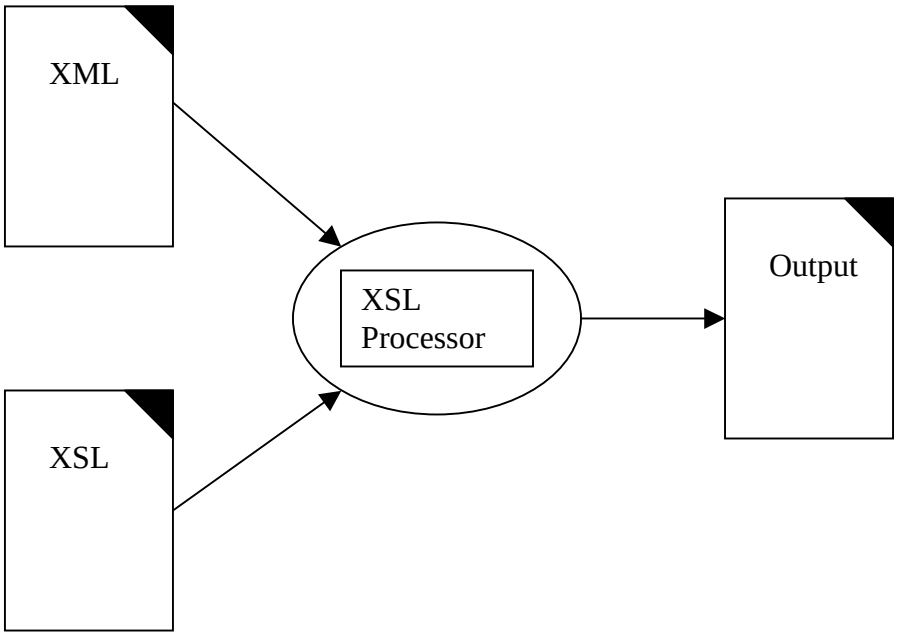


Figure 3-2. An overview diagram of an XSL Processor.

XSLT describes how to transform one XML document into another. The XSLT stylesheets are built on template structures. A template specifies what to look for in the source tree, and what to put into the result tree. The following is a code fragment from file “X3dToVml.xsl”:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
              xmlns:msxsl="urn:schemas-microsoft-com:xslt"
      xmlns:project="http://deliverable4">

<xsl:template match="X3D">
  <html xmlns:v="urn:schemas-microsoft-com:vml">
<head>
<title>Box tag from X3D to VML</title>
     <object id="VMLRender" classid="CLSID:10072CEC-8CC1-11D1-986E-
00A0C955B42E">
</object>
<style>
    v\:* {behavior: url(#VMLRender)}
</style>
</head>
<body>
<center>
<form>
 <table>
<tr>
<td></td>
<td align="center"><input type="button" value="move up"
onclick="parent.moveUp()"/></td>
<td></td>
<td></td>
<td></td>
<td align="center"><input type="button" value="rotate up"
onclick="parent.rotateUp()"/></td>
<td></td>
</tr>

<!— many tags are deleted here -->

</table>

</form>
<br/>
<br/>
<br/>
```

```
 <xsl:apply-templates/>
</center>
</body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

The above template outputs a VML document with a table at the center for each X3D tag it encounters. The table consists of seven columns. Some of the columns have buttons in them. Further templates are applied if there are any tags inside the X3D tag.

3.5 JavaScript

Extension functions written in another language such as Java or JavaScript can be embedded in an XSLT document to extend its capabilities. The common reasons for doing this are:

- To improve performance (for example when doing complex string manipulations)
- To exploit system capabilities and services
- To reuse code that already exists in another language
- For convenience, as complex algorithms and computations can be very verbose when written in XSLT

(page 132 of [KM01])

JavaScript is used in this translator for writing computational functions. The following Javascript is an extract from the translator source code.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:msxsl="urn:schemas-microsoft-com:xslt"
     xmlns:project="http://deliverable4">

<msxsl:script language="JavaScript1.2" implements-prefix="project">
<![CDATA[

  var sceneArray = new Array();

// a lot of JavaScript code is deleted
```

```
function createBox(width, height, depth)
{
  var box = new Box(width, height, depth);
  return box.toString();
}

]]>
</msxsl:script>

<!-- calling a JavaScript function -->
<xsl:template match="Box">
<!-- gets box's size from box tag -->
  <xsl:variable name="boxDim" select="@size"/>
  <xsl:variable name="x" select="substring-before($boxDim, ' ')"/>
  <xsl:variable name="rest" select="substring-after($boxDim, ' ')"/>
  <xsl:variable name="y" select="substring-before($rest, ' ')"/>
  <xsl:variable name="z" select="substring-after($rest, ' ')"/>
  <xsl:variable name="createBox" select="project:createBox($x, $y, $z)"
/>
</xsl:template>
```

To embed Javascript in an XSLT document, a namespace needs to be declared first. For example, “project” is declared as a namespace for the code above. `"JavaScript"` indicates the following script will be written in Javascript. The name for `implements-prefix` must match the namespace, in this case, it is “project”. Everything inside `<![CDATA[ ]]>` is Javascript. The above Javascript first declares an empty array called sceneArray. The function `createBox()` creates a box with specified dimension and returns a string representation for the box.

To call a function in a JavaScript section, the namespace “project” needs to be appended before the function name. The above code `"project:createBox($x, $y, $z)"` calls JavaScript function createBox().

3.6 DTD

The purpose of a DTD (Document Type Definition) is to validate an XML document. Another purpose of the DTD is to supply default attribute values. X3D allows tags

without attributes, eg <box> tag. The DTD file “X3D2VML.dtd” serves both purposes.

The following is a code fragment from file “X3D2VML.dtd”:

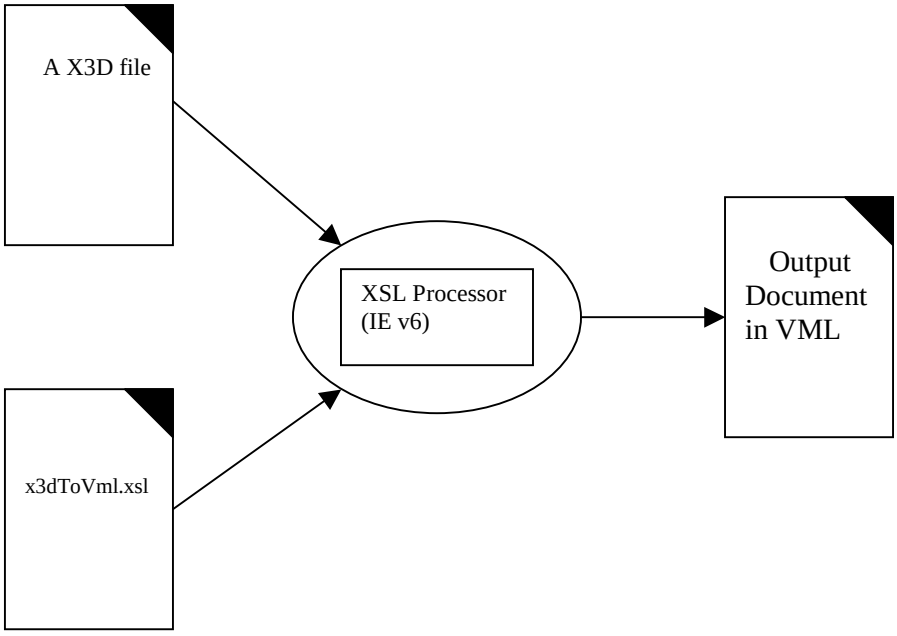
```
<!— many tags are deleted here, please see Section 5.2 to view the full
document -->

<!ENTITY % PrimitiveNodes "(Box | Cylinder | Cone | Sphere)">

<!ELEMENT Box EMPTY>
<!ATTLIST Box
  size CDATA "50 50 50">
```

The above DTD code specifies PrimitiveNodes consisting of Box, Cylinder, Cone and Sphere nodes. The default size value of a Box node is 50 50 50.

4 DESIGN

4.1 Translator Overview

Below is an overview of how my translator works.



x3d2Vml.xsl

XSLT

1. *Call appropriate JavaScript functions for each X3D tag encountered.*

JavaScript

a. Generate points in 3D for each primitive shape.
b. Transform 3D points.
c. Calculate color intensities.
d. Project points from 3D to 2D coordinates.
e. Return a very long string containing polygon information.

2. *Parse the string and add necessary VML tags/attributes to form a VML document.*

Figure 4-1. An overview diagram of the translator.

4.2 Program Design

Although the core of the translator is an XSLT style sheet, JavaScript embedded in it uses object-oriented programming. This translator has thirteen classes. The Shape class is the parent class of Box, Cylinder, Cone and Sphere. The Group and Transform classes are independent classes. The Vector, Point, Intensity, RGB, TransformationMatrix, and ProjectedPolygon classes are helper classes. The Shape class projects points in 3D to 2D using perspective projection, computes normal vector for each polygon, and returns shape polygons in 2D in a very long string. The Box, Cylinder, Cone, and Sphere classes generate points for a box, cylinder, cone, and sphere respectively. The Vector class provides simple vector operations, such as normalization, dot product, and magnitude computation. The Point class performs transformation specified in a transformation matrix. The Intensity class calculates intensity using the Phong lighting model. The RGB class converts a color in rgb to hsv, sets saturation to intensity, and then converts hsv back to rgb. The Group class groups Shape Group and Transform tags together. The

Transform class supports rotation, translation, and scaling. The TransformationMatrix is a 4X4 matrix which contains information to transform a point using rotation, scaling, translation, and projection. The ProjectedPolygon class sorts polygons from farthest to nearest.

Below is an UML diagram of all thirteen classes.

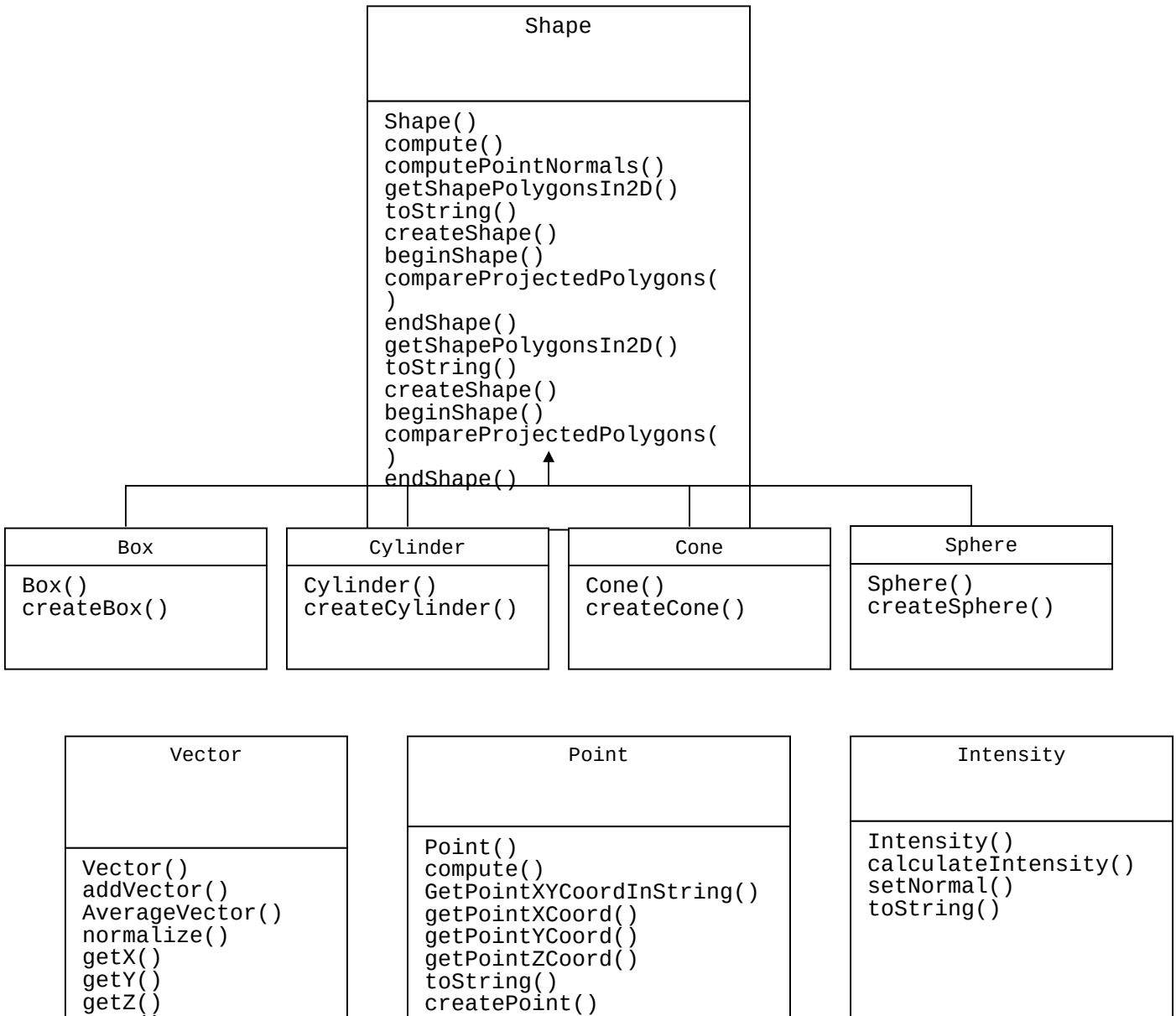


Figure 4-2. UML Diagram of classes.

Please refer to Appendix A for description of all class member variables and functions for each class.

5. REQUIREMENTS

Five things are required to run the translator. They are listed from Section 5.1 to Section 5.5.

5.1 An X3D Input Document

First of all, an X3D input document is required. Below is a sample X3D input file.

A box that has width of 200.0, height of 300.0, and depth of 400.0 in X3D:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="X3d2Vml.xsl"?>
<!DOCTYPE X3D SYSTEM " X3D2VML.dtd">
<X3D>
      <Scene>
            <Shape>
                  <Appearance><Material/></Appearance>
                  <Box size="200.0 300.0 400.0" />
            </Shape>
      </Scene>
</X3D>
```

5.1.1 X3D Tags the Translator Supported

As mentioned in the previous section, the translator supports primitive tags as well as grouping tags. Below is a listing all X3D tags and attributes supported by the translator. The default attribute values are defined in the file X3dToVml.dtd.

- X3D
- Scene
- Group DEF
  USE
- Transform translation “0 0 0”
  rotation “1 0 0 0”

	scale	“1 1 1”

- Shape
- Appearance

• Box	size	“50 50 50”
• Cone	bottomRadius	“50”
	Height	“100”
• Cylinders	height	“100”
	Radius	“50”
• Sphere	radius	“50”

The tags are in the first column of the above list. The attributes are in second column. The last column are attribute default values.

5.2 X3dToVml.dtd

“X3dToVml.dtd” validates the input X3D document and supplies default attribute values.

The following is the “X3D2VML.dtd” document.

```
<!ENTITY % PrimitiveNodes "(Box | Cylinder | Cone | Sphere)">
<!ENTITY % GroupingNodes  "(Group | Transform)">
<!ENTITY % SceneNodes     "(%GroupingNodes;)*, (Shape)*">
<!ENTITY % ChildNodes  "(%SceneNodes;)*">

<!ELEMENT X3D (Scene)>
<!ELEMENT Scene
  (%SceneNodes;)>

<!ELEMENT Group %ChildNodes;>

<!ELEMENT Transform %ChildNodes;>
<!ATTLIST Transform
  translation CDATA "0 0 0"
  scale       CDATA "1 1 1"
  rotation    CDATA "1 0 0 0">

<!ELEMENT Shape
  (Appearance?, (%PrimitiveNodes;)?)>

<!ELEMENT Appearance (Material)>

<!ELEMENT Material EMPTY>
<!ATTLIST Material
  diffuseColor     CDATA "0.0 0.0 1.0">

<!ELEMENT Box EMPTY>
<!ATTLIST Box
  size CDATA "50 50 50">
```

```
<!ELEMENT Cylinder EMPTY>
<!ATTLIST Cylinder
  height CDATA "100"
  radius CDATA "50">

<!ELEMENT Cone EMPTY>
<!ATTLIST Cone
  height CDATA "100"
  bottomRadius CDATA "50">

<!ELEMENT Sphere EMPTY>
<!ATTLIST Sphere
  radius CDATA "50">
```

The basic idea of the above DTD is that an X3D node must be the root node of all X3D documents. A Scene node must be the only child node of the X3D node. A scene node may contain a grouping node or a shape node. A Grouping node consists of the Group and the Transform node. A Grouping node could have a Shape node and/or nested Grouping node. A shape node could have a Box, Cylinder, Cone, or Sphere. The default size attribute of a Box node is 50 50 50. The default height of a Cylinder node is 100 and the radius is 50. The default height of a Cone node is 100 and the bottomRadius is 50. The default radius of a Sphere node is 50. The color of a shape is specified by its child Appearance node. Its default color is blue.

5.3 X3dToVml.xsl

The X3dToVml.xsl is the main part of the translator. It is an XSLT style sheet, which transforms an X3D document into a VML document. The input X3D document needs to specify X3dToVml.dtd as its DTD and X3dToVml.xsl as its style sheet. An example of an X3D input document is shown in Section 5.1. Section 6.1 will explain X3dToVml.xsl in great detail.

5.4 X3dToVml.html

X3dToVml.html expands USE attributes and contains function to handle button click events. The translator works correctly without the X3dToVml.html file if the input document does not have any USE attributes and only render the input document once (no button clicks). The X3dToVml.html has functions to handle each button click event and to refresh the screen with a new perspective. Section 6.2 will explain this file in detail.

5.5 Web Browser

IE (Internet Explorer) 5 and later natively support VML. But IE5.0 and 5.5 do not render VML code generated by my translator correctly. They only output the VML code on the screen but do not render the VML graphics. The VML code is helpful for the analyst for debugging, but not helpful for the users. However, the IE 6 works correctly. It draws the VML graphics.

6. IMPLEMENTATION

6.1 X3dToVml.xsl

X3dToVml.xsl is the main part of the translator. The rest of this section will explain it in detail.

6.1.1 Getting Attribute Values

X3dToVml.xsl consists of two parts, the template matching and the JavaScript as mentioned. One of the tasks for some templates is to get attribute values, for instance, Box and Transform templates. Those attributes having multiple values need to be handled separately. The space is used as a delimiter. Attribute values are obtained one by one. The following code is for parsing attribute values for a Box tag:

```
<xsl:template match="Box">
  <!-- gets box's size from box tag -->
  <xsl:variable name="boxDim" select="@size"/>
  <xsl:variable name="x" select="substring-before($boxDim, ' ')"/>
  <xsl:variable name="rest" select="substring-after($boxDim, ' ')"/>
  <xsl:variable name="y" select="substring-before($rest, ' ')"/>
  <xsl:variable name="z" select="substring-after($rest, ' ')"/>
  <xsl:variable name="createBox" select="project:createBox($x, $y, $z)"
/>
</xsl:template>
```

The above code first selects size attribute value of the <Box> tag and saves it into variable “boxDim”. Then it gets x, y, and z values one at a time using space as the delimiter. Last, it calls createBox() function with x, y, and z variables as parameters.

6.1.2 Creating Primitive Shapes

This section describes the primitive shapes available in X3D and how the algorithm represents them. The primitive shapes are represented by points. To begin with a box tag, it requires width, height, and depth to specify its dimensions. From its dimensions, the eight corners can be found. The box below is formed by points 0 – 7.

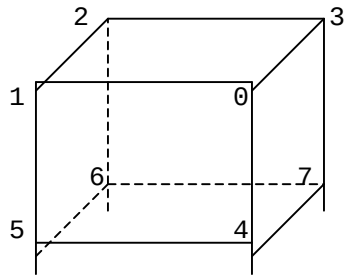


Figure 6-1. A box is represented by eight points.

These eight points are organized into a 2X4 array. The Points 0, 1, 2, and 3 form row0, and the Points 4, 5, 6, and 7 form row1. So the points of a box represented by a 2X4 array are as follows:

Box array:

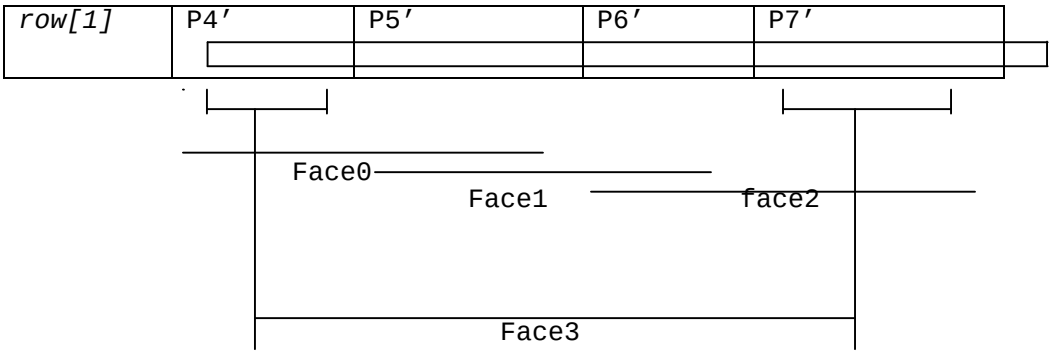
	Column[0]	*Column[1]*	*Column[2]*	*Column[3]*
row[0]	P0	P1	P2	P3
row[1]	P4	P5	P6	P7

After organizing the eight points into these arrays, a transformation if any is applied. Then the points are projected from 3D to 2D, and the intensity is calculated. The six faces of a box are easily represented as follows:

Box array':

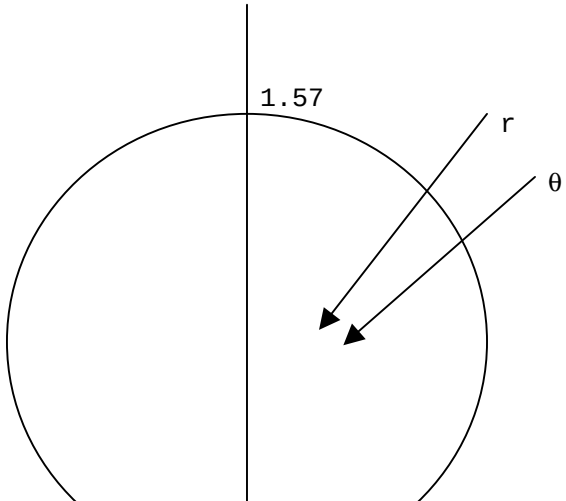
	Column[0]	*Column[1]*	*Column[2]*	*Column[3]*
row[0]	P0'	P1'	P2'	P3'

face5
(top)
face6
(bottom)



Let's move on to how to handle the cylinder primitive. This primitive has two attributes, radius and height. The cylinder is represented by a two dimensional array. There are 24 points in each row. All points on the top are in one array and all points on the bottom are in another array.

The primitive shapes are centered at the origin by default in X3D. The height of a cylinder's top is height/2, its bottom is –height/2. The y value stays the same for each row, and only the x and the z values change for each point. The x and z values can be calculated using sine and cosine.



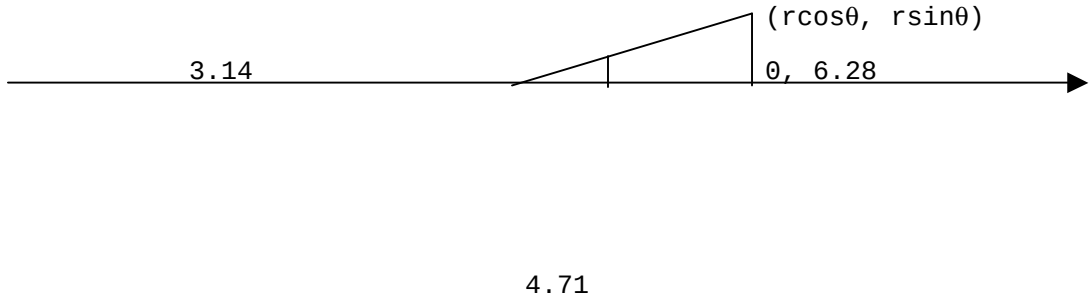


Figure 6-2. Calculating X and Z values of a point on a cylinder surface.

The points on the first row of a cylinder are created using a for loop:

```
var y = height / 2;
var angle = Math.PI / 12;
var cylinderArray0 = new Array();

for (var i = 0; i < Math.PI * 2; i += angle)
{
  cylinderArray0.push(new Point(radius * Math.cos(i), y, radius * Math.sin(i)));
}
```

The above source code creates and initializes variables `y`, `angle`, and `cylinderArray0`. The points on the top face are created and pushed into `cylinderArray0`.

Next, let's move on to describe a cone. The height and bottomRadius for a cone are specified in X3D. There are 24 point on each row as for a cylinder. The y value (how many levels need to be drawn) and radius have be obtained at each height. There are two global variables bigHeightIncrement and smallHeightIncrement with default values 20 and 5 respectively. The BigHeightIncrement variable is used for cones with height greater than or equal to 100. The smallHeightIncrement variable is used for cones with height less than 100. So, tall cones are drawn with more levels, and short cones are drawn with fewer levels. After knowing how many levels are to be drawn, the next question is to find the radius at each level. This is not hard because the height and the radius of a cone are proportional.

Figure 4-3 is a side view of a cone with radius of 100 and height of 200. To find the radius at a height of 160, multiply 100 / 200 by 160, which is the ratio of the radius over height.

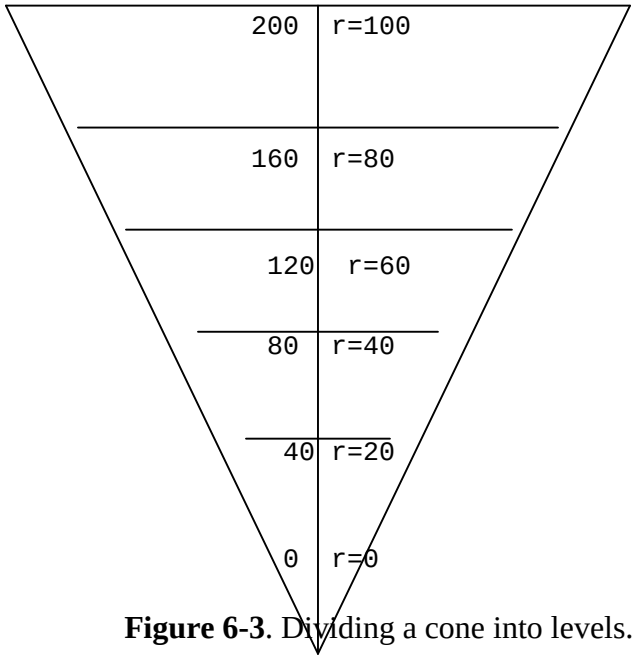


Figure 6-3. Dividing a cone into levels.

At each level, the number of points are the same. They are calculated the same way as was done for a cylinder.

The sphere primitive has only a radius attribute. The overall height is two times the radius. If the overall height is greater than or equal to 100, the overall height is divided by the bigHeightIncrement variable otherwise it is divided by the smallHeightIncrement variable. The following is an example of a sphere divided into multiple levels:

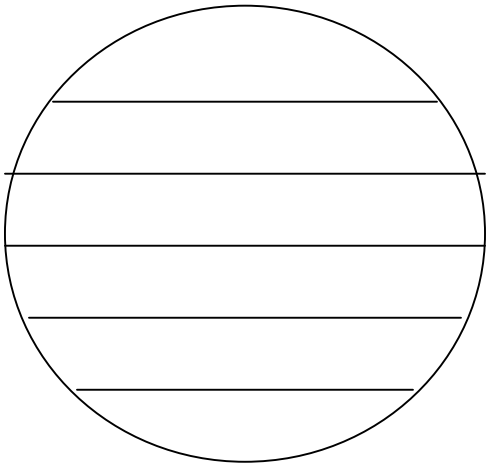
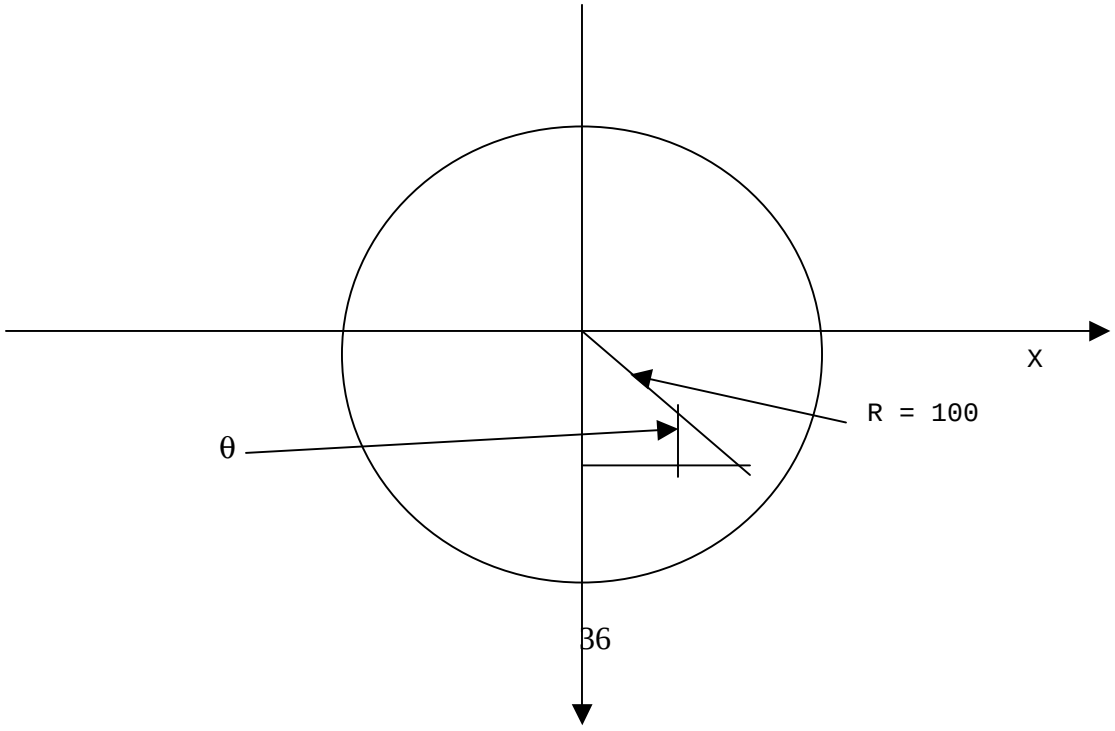


Figure 6-4. A Sphere is dividing into multiple levels.

Now, we have to find the radius (x value) for each level. All radii going through the origin are constant. For each level, the θ can be obtained using the formula $\theta = \sin^{-1}(\text{height} / \text{sphere radius})$. After obtaining θ, one can find the x value for that level, which is r = sphere radius * cos θ. Figure 6-5 shows a side view of a sphere with radius 100.



Y

Figure 6-5. Calculating radius at some height level of a sphere.

The table below shows the x values at different levels for a sphere with radius 100.

height	θ sin^{-1}*(height / sphere radius)*	*r* *sphere radius * cos* θ
-100	1.570	0
-80	0.927	60
-60	0.644	80
-40	0.412	91.65
-20	0.201	97.9
0	0	100
20	0.201	97.9
40	0.412	91.65
60	0.644	80
80	0.927	60
100	1.570	0

Table 6-1. Table of x values for a sphere with radius 100.

After getting the x value at each level, points at each level are calculated the same way as was done for a cone or cylinder.

6.1.3 The Phong Lighting Model

The material for this section is based on the book [SRB02]. The Phong lighting model is used to calculate light intensity for a point on a surface in a realistic way. In this project, one light source is used. This is the source coming from the user's headlight. The following formula is used to calculate intensity:

$$I = 0.4 + 0.3(l \bullet n) + 0.3(h \bullet n)^2$$

The first term 0.4 represents ambient lighting, which is the light that shines evenly on all the surfaces and in all directions. It is the background light. The ambient lighting

produces a flat shading for each surface. The scenes are usually not rendered with ambient light by itself.

The second term $0.3(l \bullet n)$ is used to calculate diffuse lighting, which is constant over each surface in a scene. A surface perpendicular to the direction of the light source is more illuminated than an equal-size surface at some angle to the light source. The *l* factor is a point light source unit vector. The *n* factor is a surface normal vector.

The third term $0.3(h \bullet n)^2$ calculates the specular lighting, which is the bright spot visible from certain viewing directions. The *h* factor is the half way vector. The half way vector is the unit vector half way between the light source direction and the view direction. The *n* factor is a surface normal vector.

6.1.4 Gouraud Shading

There are many ways to render polygon surfaces, namely, flat shading, Gouraud shading, and Phong shading. Flat shading is a fast and simple method for rendering polygons with constant intensity. It produces a dull shading effect. Phong shading is an accurate rendering method. It first interpolates normal vectors, then apply an illumination model to each surface point [HB97]. VML has a gradient tag, which may specify up to two colors for gradient painting. Phong shading is not used in this project because VML's native tag could not be used for shading. Gouraud shading renders a polygon surface by linearly interpolating intensity values across the surface [HB97]. It is chosen because it is fast, simple, and also provides reasonably accurate surface shading. First, find the two

intensities for each polygon as mentioned in the introduction, then find two colors corresponding to the two intensities, and finally use VML's gradient tag for shading.

There are five steps in shading. First, find a normal vector for each polygon and record this vector for each point in this polygon. The source code below works for all primitives by calculating normals for the polygons in it. It adds normal vector as it loops through.

```
Shape.prototype.computePointNormals = function()
{
  var string = "";

  var x1;
  var y1;
  var z1;

  var x2;
  var y2;
  var z2;

  var x3;
  var y3;
  var z3;

  var A;
  var B;
  var C;

  for (var i = this.shapeArray.length - 1; i >= 1; --i)
  {
    for (var j = this.shapeArray[i].length - 1; j >= 1; --j)
    {
       x1 = this.shapeArray[i][j].getPointXCoord();
            y1 = this.shapeArray[i][j].getPointYCoord();
       z1 = this.shapeArray[i][j].getPointZCoord();

       x2 = this.shapeArray[i][j-1].getPointXCoord();
       y2 = this.shapeArray[i][j-1].getPointYCoord();
       z2 = this.shapeArray[i][j-1].getPointZCoord();

       x3 = this.shapeArray[i-1][j-1].getPointXCoord();
       y3 = this.shapeArray[i-1][j-1].getPointYCoord();
       z3 = this.shapeArray[i-1][j-1].getPointZCoord();

       A = y1*(z2 - z3) + y2*(z3 - z1) + y3*(z1 - z2);
       B = z1*(x2 - x3) + z2*(x3 - x1) + z3*(x1 - x2);
       C = x1*(y2 - y3) + x2*(y3 - y1) + x3*(y1 - y2);

       //record normals into these four points
       this.shapeArray[i][j].normal.addVector(A, B, C);
       this.shapeArray[i][j-1].normal.addVector(A, B, C);
       this.shapeArray[i-1][j-1].normal.addVector(A, B, C);
       this.shapeArray[i-1][j].normal.addVector(A, B, C);
    }

    // calculate normals of a polygon forming by first and last 4 points
    x1 = this.shapeArray[i][0].getPointXCoord();
    y1 = this.shapeArray[i][0].getPointYCoord();
    z1 = this.shapeArray[i][0].getPointZCoord();

    x2 = this.shapeArray[i][this.shapeArray[i].length-1].getPointXCoord();
    y2 = this.shapeArray[i][this.shapeArray[i].length-1].getPointYCoord();
```

```
      z2 = this.shapeArray[i][this.shapeArray[i].length-1].getPointZCoord();


      x3 = this.shapeArray[i-1][this.shapeArray[i].length-1].getPointXCoord();
      y3 = this.shapeArray[i-1][this.shapeArray[i].length-1].getPointYCoord();
      z3 = this.shapeArray[i-1][this.shapeArray[i].length-1].getPointZCoord();

      A = y1*(z2 - z3) + y2*(z3 - z1) + y3*(z1 - z2);
      B = z1*(x2 - x3) + z2*(x3 - x1) + z3*(x1 - x2);
      C = x1*(y2 - y3) + x2*(y3 - y1) + x3*(y1 - y2);

     //record normals into these four points
     this.shapeArray[i][0].normal.addVector(A, B, C);
     this.shapeArray[i][this.shapeArray[i].length-1].normal.addVector(A, B, C);
     this.shapeArray[i-1][this.shapeArray[i].length-1].normal.addVector(A, B, C);
     this.shapeArray[i-1][0].normal.addVector(A, B, C);

    }

  }
```

Second, average and normalize all polygon normals adjacent to each point. This is done by calling normal's averageVector() method. For instance, to average normal vectors for this shape array, call this shape array's normal variable's member method averageVector() as follows:

```
this.shapeArray[i][j].normal.averageVector();
```

Third, get two normals to represent each polygon. Since VML's gradient tag only supports up to two colors, average the sum of two vertical normals together to have total of two normals for each polygon.

```
this.normal1.addVector(this.shapeArray[i][j].normal.getX(),
                       this.shapeArray[i][j].normal.getY(),
                       this.shapeArray[i][j].normal.getZ()
                      );

this.normal1.addVector(this.shapeArray[i][j-1].normal.getX(),
                       this.shapeArray[i][j-1].normal.getY(),
                      this.shapeArray[i][j-1].normal.getZ()
                      );

this.normal2.addVector(this.shapeArray[i-1][j].normal.getX(),
                       this.shapeArray[i-1][j].normal.getY(),
                       this.shapeArray[i-1][j].normal.getZ()
                      );

this.normal2.addVector(this.shapeArray[i-1][j-1].normal.getX(),
                       this.shapeArray[i-1][j-1].normal.getY(),
                       this.shapeArray[i-1][j-1].normal.getZ()
                     );

this.normal1.averageVector();
this.normal2.averageVector();
```

After getting two normals for each polygon, find their intensities. For instance, to find intensity for normal1, just declare an intensity variable and call its `calculateIntensity()` member method, which takes a normal vector as parameter.

```
var intensity1 = new Intensity();


I1 = intensity1.calculateIntensity(this.normal1);
```

The Phong lighting model is used as the illumination model here.

Fourth, after getting two intensities for each polygon, find their colors. Since HSV is a more natural color model, first convert rgb to hsv, set saturation to intensity, then convert hsv to rgb. This is done by calling RGB's member method rgbToHsvToRgb(), which takes intensity as parameter. The following source code is an example of calling that method:

```
var rgbTemp = new RGB(this.rgb.getR(), this.rgb.getG(), this.rgb.getB());
rgbTemp.rgbToHsvToRgb(I1);
```

Fifth, use VML's gradient tag to shade a polygon. The following is a sample source code of a polygon after adding a shading tag in VML:

```
<v:polyline
   points="-19.016395188814574pt,
           0pt,
           30.54242809473926pt,
           0pt,
           30.02064503545629pt,
          -16.911293966990907pt,
          -18.693756532647207pt,
          -16.79494671277145pt,
          -19.016395188814574pt,
          0pt
         "
   fillcolor="rgb(58, 58, 255)">
<v:stroke on="false"/>
<v:fill
   method="linear sigma"
   angle="180"
   type="gradient"
   color2="rgb(62, 62,255)"/>
</v:polyline>
```

The above VML code draws a polygon with four edges. The polygon is rendered with linear sigma gradient shading.

6.1.5 Transformation

Now, let's see how to implement transformations. The <Transformation> tag has three attributes: translation, rotation, and scaling. Assign default values if these attributes are not specified. A 4X4 matrix is used to represent a transformation. A 4X4 matrix is shown below.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Figure 6-6. A 4X4 Matrix.

Sections 5.1 to 5.3 will explain how to use a 4X4 matrix to implement translation, scaling and rotation.

6.1.5.1 Translation

The objects are centered at the world coordinate system's origin by default. The translation attribute of the <Transform> tag allows objects to move in the x, y, and z directions. The elements at positions $a_{03,}$ $a_{13\ and}$ a_{23} in Figure 6-6 are used for translation. To move a shape 20 units left, 50 units up and 30 units forward, first create a 4X4 matrix representing these movements (an identity matrix and with 20, 50, and 30 at position $a_{03,}$ $a_{13\ and}$ a_{23} respectively), and then multiply a matrix representing the current state of the shape.

$$\begin{bmatrix} 1 & 0 & 0 & 20 \\ 0 & 1 & 0 & 50 \\ 0 & 0 & 1 & 30 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 6-7. Translating a Shape.

In Figure 5-7, the matrix represents translating 20 units left, 50 units up and 30 units forward. The inverse of the translation matrix is obtained by negating the translation distances. This produces a translation in the opposite direction.

The code for the translation function is shown below.

```
TransformationMatrix.prototype.setMatrixTranslation = function(x, y, z)
{
        var temp = new TransformationMatrix();

        temp.setIdentity();
        temp.matrix[0][3] = x;
        temp.matrix[1][3] = y;
        temp.matrix[2][3] = z;

        this.matrixMultiply(temp);
}
```

6.1.5.2 Rotation

Another attribute of the <Transform> tag is the rotation attribute. This attribute takes four values, vector (x, y, z) and rotation angle θ in radians. The vector (x, y, z) represents the axis about which the rotation will be carried out. The elements at positions $a_{00}, a_{01}, a_{02}, a_{10}, a_{11}, a_{12}, a_{20}, a_{21}, a_{22}$ in Figure 6-8 are used for rotation.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The calculation for each element is as follows:

$$a_{00} = 1.0 - 2\left(\frac{\sin(\theta/2.0)y}{|x+y+z|}\right)^2 - 2\left(\frac{\sin(\theta/2.0)z}{|x+y+z|}\right)^2$$

$$a_{01} = 2\left(\frac{\sin(\theta/2.0)x}{|x+y+z|}\right)\left(\frac{\sin(\theta/2.0)y}{|x+y+z|}\right) - 2\cos(\theta/2.0)\left(\frac{\sin(\theta/2.0)z}{|x+y+z|}\right)$$

$$a_{02} = 2\left(\frac{\sin(\theta/2.0)x}{|x+y+z|}\right)\left(\frac{\sin(\theta/2.0)z}{|x+y+z|}\right) + 2\cos(\theta/2.0)\left(\frac{\sin(\theta/2.0)b}{|x+y+z|}\right)$$

$$a_{10} = 2\left(\frac{\sin(\theta/2.0)x}{|x+y+z|}\right)\left(\frac{\sin(\theta/2.0)y}{|x+y+z|}\right) + 2\cos(\theta/2.0)\left(\frac{\sin(\theta/2.0)z}{|x+y+z|}\right)$$

$$a_{11} = 1.0 - 2\left(\frac{\sin(\theta/2.0)x}{|x+y+z|}\right)^2 - 2\left(\frac{\sin(\theta/2.0)z}{|x+y+z|}\right)^2$$

$$a_{12} = 2\left(\frac{\sin(\theta/2.0)y}{|x+y+z|}\right)\left(\frac{\sin(\theta/2.0)z}{|x+y+z|}\right) - 2\cos(\theta/2.0)\left(\frac{\sin(\theta/2.0)x}{|x+y+z|}\right)$$

$$a_{20} = 2\left(\frac{\sin(\theta/2.0)x}{|x+y+z|}\right)\left(\frac{\sin(\theta/2.0)z}{|x+y+z|}\right) - 2\cos(\theta/2.0)\left(\frac{\sin(\theta/2.0)y}{|x+y+z|}\right)$$

$$a_{21} = 2\left(\frac{\sin(\theta/2.0)y}{|x+y+z|}\right)\left(\frac{\sin(\theta/2.0)z}{|x+y+z|}\right) + 2\cos(\theta/2.0)\left(\frac{\sin(\theta/2.0)x}{|x+y+z|}\right)$$

$$a_{22} = 1.0 - 2\left(\frac{\sin(\theta/2.0)x}{|x+y+z|}\right)^2 - 2\left(\frac{\sin(\theta/2.0)y}{|x+y+z|}\right)^2$$

Figure 6-8. Rotating a Shape.

The code for the rotation function is shown below.

```
TransformationMatrix.prototype.setMatrixRotation = function(x, y, z, radianAngle)
                                                   {
  var length = Math.sqrt(x*x + y*y + z*z);
  var cosA2 = Math.cos(radianAngle / 2);
  var sinA2 = Math.sin(radianAngle / 2);
  var a = sinA2 * x / length;
  var b = sinA2 * y / length;
  var c = sinA2 * z / length;
  var temp = new TransformationMatrix();

  temp.setIdentity();
  temp.matrix[0][0] = 1 - 2*b*b - 2*c*c;
  temp.matrix[0][1] = 2*a*b - 2*cosA2*c;
  temp.matrix[0][2] = 2*a*c + 2*cosA2*b;

  temp.matrix[1][0] = 2*a*b + 2*cosA2*c;
  temp.matrix[1][1] = 1 - 2*a*a - 2*c*c;
  temp.matrix[1][2] = 2*b*c - 2*cosA2*a;

  temp.matrix[2][0] = 2*a*c - 2*cosA2*b;
  temp.matrix[2][1] = 2*b*c + 2*cosA2*a;
  temp.matrix[2][2] = 1 - 2*a*a - 2*b*b;

  this.matrixMultiply(temp);
}
```

6.1.5.3 Scaling

Next, let's look at the scaling attribute of the <Transform> tag. The objects can be enlarged and shrunk by manipulating the scaling attribute. The scaling attribute has three values, which specify scaling in the x, y, and z axes. The elements at positions $a_{00,}$ a_{11} and a_{22} in Figure 6-6 are used for scaling. In order to scale a shape, a similar idea can be used as with translation. For example, stretching a shape 1.3 along the x-axis, 0.7 along the y-axis, and 2 in the z-axis can be presented as follows:

$$\begin{bmatrix} 1.3 & 0 & 0 & 0 \\ 0 & 0.7 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 6-9. Scaling a Shape.

The relative dimensions in the object are changed in this example because the scaling factors are not all equal. The inverse scaling matrix is formed by replacing the scaling factors with their reciprocals. The code for the scaling function is shown below.

```
TransformationMatrix.prototype.setMatrixScaling = function(x, y, z)
{
	var temp = new TransformationMatrix();

	temp.setIdentity();
	temp.matrix[0][0] = x;
	temp.matrix[1][1] = y;
	temp.matrix[2][2] = z;

	this.matrixMultiply(temp);
}
```

6.1.6 Grouping

Section 6.1.6 explains how to transform shapes. The grouping tags allow manipulation of a group of shapes together. The <Group> and <Transform> tags are grouping tags. An array of transformation matrices called transformationMatrixArray is used to implement grouping. Both the tags, namely, <Group> and <Transform> tags, are used for grouping. When the XML processor encounters a Transform tag, it creates a Transformation matrix and appends it to the transformationMatrixArray. Nothing is appended to the transformationMatrixArray if a Group tag is encountered, because it does not change anything. When a Shape node is encountered, the transformation matrices in the transformationMatrixArray are multiplied and the product is stored in the Shape member variable transformationMatrix, which represents the transformation state of the shape.

When a Transform end tag is encountered, the last transform matrix in the transformationMatrixArray is popped off. Nothing is removed from the

transformationMatrixArray when a Group end tag is encountered, because nothing is appended for a Group beginning tag.

6.1.7 Parsing 2D Polygon String

XSLT is not a DOM document, so the DOM's document.write() function could not be used in JavaScript to output VML tags directly. However, the JavaScript functions can return strings. This is done by concatenating all information required to create VML tags into a very long string and parsing it later. The Template "parse-polygonString" is used for the parsing process.

```
<!-- parses a polygon's string (either a Box, Cone, Cylinder, or Sphere)
-->
<xsl:template name="parse-polygonString">
  <xsl:param name="paramString"/>
  <xsl:variable name="normalizedParam" select="concat(normalize-
space($paramString), ' ')"/>


  <xsl:variable name="x0" select="substring-before($normalizedParam, '
')"/>
  <xsl:variable name="rest1" select="substring-after($normalizedParam, '
')"/>
  <xsl:variable name="y0" select="substring-before($rest1, ' ')"/>
  <xsl:variable name="rest2" select="substring-after($rest1, ' ')"/>
  <xsl:variable name="x1" select="substring-before($rest2, ' ')"/>
  <xsl:variable name="rest3" select="substring-after($rest2, ' ')"/>
  <xsl:variable name="y1" select="substring-before($rest3, ' ')"/>
  <xsl:variable name="rest4" select="substring-after($rest3, ' ')"/>
  <xsl:variable name="x2" select="substring-before($rest4, ' ')"/>
  <xsl:variable name="rest5" select="substring-after($rest4, ' ')"/>
  <xsl:variable name="y2" select="substring-before($rest5, ' ')"/>
  <xsl:variable name="rest6" select="substring-after($rest5, ' ')"/>
  <xsl:variable name="x3" select="substring-before($rest6, ' ')"/>
  <xsl:variable name="rest7" select="substring-after($rest6, ' ')"/>
  <xsl:variable name="y3" select="substring-before($rest7, ' ')"/>
  <xsl:variable name="rest8" select="substring-after($rest7, ' ')"/>

  <xsl:variable name="r1" select="substring-before($rest8, ' ')"/>
  <xsl:variable name="rest9" select="substring-after($rest8, ' ')"/>
  <xsl:variable name="g1" select="substring-before($rest9, ' ')"/>
  <xsl:variable name="rest10" select="substring-after($rest9, ' ')"/>
  <xsl:variable name="b1" select="substring-before($rest10, ' ')"/>
  <xsl:variable name="rest11" select="substring-after($rest10, ' ')"/>
  <xsl:variable name="r2" select="substring-before($rest11, ' ')"/>
```

```
  <xsl:variable name="rest12" select="substring-after($rest11, ' ')"/>
  <xsl:variable name="g2" select="substring-before($rest12, ' ')"/>
  <xsl:variable name="rest13" select="substring-after($rest12, ' ')"/>
  <xsl:variable name="b2" select="substring-before($rest13, ' ')"/>
  <xsl:variable name="rest14" select="substring-after($rest13, ' ')"/>

  <xsl:call-template name="drawPolygon">
    <xsl:with-param name="x0" select="$x0"/>
    <xsl:with-param name="y0" select="$y0"/>
    <xsl:with-param name="x1" select="$x1"/>
    <xsl:with-param name="y1" select="$y1"/>
    <xsl:with-param name="x2" select="$x2"/>
    <xsl:with-param name="y2" select="$y2"/>
    <xsl:with-param name="x3" select="$x3"/>
    <xsl:with-param name="y3" select="$y3"/>
    <xsl:with-param name="r1" select="$r1"/>
    <xsl:with-param name="g1" select="$g1"/>
    <xsl:with-param name="b1" select="$b1"/>
    <xsl:with-param name="r2" select="$r2"/>
            <xsl:with-param name="g2" select="$g2"/>
    <xsl:with-param name="b2" select="$b2"/>
  </xsl:call-template>

  <!--- call recursively for $rest14 -->
  <xsl:variable name="length" select="string-length($rest14)"/>
  <xsl:choose>
    <xsl:when test="$length > 0">
      <xsl:call-template name="parse-polygonString">
      <xsl:with-param name="paramString" select="$rest14"/>
      </xsl:call-template>
    </xsl:when>
  </xsl:choose>

</xsl:template>
```

First, the template “parse-polygonString” gets a very long string from parameter “paraString”. Then it normalizes space and concatenates the parameter string so that extra spaces in between are eliminated. Later, it extracts out four points and two colors in rgb. After that, it calls the “drawPolygon” template with these four points and two colors as the parameters. Finally, it calls the “parse-polygonString” template recursively if the remaining string is not empty.

6.1.8 Generating VML tags

The top and bottom part of a VML document is generated when an X3D is encountered. Following are the text nodes the X3D template inserts into the result tree:

```
<html xmlns:v="urn:schemas-microsoft-com:vml"
xmlns:o="urn:schemas-microsoft-com:office:office"
xmlns="http://www.w3.org/TR/REC-html40">

<head>
<style>
v\:* {behavior:url(#default#VML);}
</style>
</head>

<body>

<!—many polyline tags generated by drawPolygon template, which is called
by other templates  -->

</body>

</html>
```

The html root element binds the namespace prefix v to the URI `urn:schemas-microsoft-com:vml`. The head element contains a style element which indicates that VML's default renderer will be used for display. The VML renderer is a program installed with Internet Explorer 5 and later [HR01]. The Polyline tag and fill tags are the core tags used for the translator. The Polyline tag is a series of straight lines drawn between successive pairs of points [HR01]. The Fill tag specifies how and with what the shape is to be filled. Gradient filling is used here.

Generating the polyline tags is tricky because the XSLT processor does not allow one to create a tag like `<v:polylinepoints="181pt, 154pt, 126pt, 140pt, 126pt, 180pt, 181pt, 214pt, 181pt, 154pt"/>` directly for two reasons. The first reason is that "<" has a special meaning and "v:" means a namespace to the XSLT processor. The second reason is that the XSLT processor does not allow attributes to have multiple

values. What needs to be done is to output polyline tag in many steps. A polyline is broken down into many XSLT tags. Please see the source code below on how to do it:

```
<xsl:template name="drawPolygon">
  <xsl:param name="x0"/>
  <xsl:param name="y0"/>
  <xsl:param name="x1"/>
  <xsl:param name="y1"/>
  <xsl:param name="x2"/>
  <xsl:param name="y2"/>
  <xsl:param name="x3"/>
  <xsl:param name="y3"/>
  <xsl:param name="r1"/>
  <xsl:param name="g1"/>
  <xsl:param name="b1"/>
  <xsl:param name="r2"/>
  <xsl:param name="g2"/>
  <xsl:param name="b2"/>


<xsl:text disable-output-escaping="yes">&lt;v:polyline points="</xsl:text>
  <xsl:value-of select="$x0"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y0"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$x1"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y1"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$x2"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y2"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$x3"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y3"/>pt,
  <xsl:value-of select="$x0"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y0"/>pt
  <xsl:text disable-output-escaping="yes">" fillcolor="rgb(</xsl:text>
  <xsl:value-of select="$r1"/>,
  <xsl:value-of select="$g1"/>,
  <xsl:value-of select="$b1"/>
  <xsl:text disable-output-escaping="yes">)"&gt;</xsl:text>
  <xsl:text disable-output-escaping="yes">&lt;v:stroke on="false"/></xsl:text>
  <xsl:text disable-output-escaping="yes">&lt;v:fill method="linear sigma" angle="180"
type="gradient" color2="rgb(</xsl:text>
  <xsl:value-of select="$r2"/>,
  <xsl:value-of select="$g2"/>,
  <xsl:value-of select="$b2"/>
  <xsl:text disable-output-escaping="yes">)"/&gt;</xsl:text>
  <xsl:text disable-output-escaping="yes">&lt;/v:polyline&gt;</xsl:text>

</xsl:template>
```

A sample polygon tag generated by the “drawPolygon” template is shown below:

```
<v:polyline
   points="-19.016395188814574pt,
           0pt,
           30.54242809473926pt,
           0pt,
           30.02064503545629pt,
          -16.911293966990907pt,
          -18.693756532647207pt,
          -16.79494671277145pt,
          -19.016395188814574pt,
          0pt
```

```
        "
   fillcolor="rgb(58, 58, 255)">
<v:stroke on="false"/>
<v:fill
   method="linear sigma"
   angle="180"
   type="gradient"
   color2="rgb(62, 62,255)"/>
</v:polyline>
```

The "v" is the namespace prefix which binds to the URI `urn:schemas-microsoft-com:vml`. The Polyline tags draw four edges and renders with linear sigma gradient shading vertically with rgb colors "58, 58, 255" and "62, 62,255" in 0 to 256 color scale.

6.2 X3dToVml.html

X3dToVml.html is a frameset. The frameset contains JavaScript code and one frame called "vmlFrame". The frameset and frame structure is used here to refresh with a new VML scene every time a button is clicked.

6.2.1 Loading An X3D Input Document

Microsoft ActiveXObjects are used to load both the X3D input document and the XSLT stylesheet.

```
var xml = new ActiveXObject("Microsoft.XMLDOM");   // Load XML
xml.async = false;
xml.load("cylinder.xml");

var xsl = new ActiveXObject("Microsoft.XMLDOM");   // Load the XSL
xsl.async = false;
xsl.load("x3d2vml.xsl");
```

The X3D input document and the XSLT stylesheet are loaded and stored in variables xml and xsl respectively.

6.2.1.1 Handling DEF and USE

A Group can be defined using the DEF attribute and can be used later using the USE attribute. The DEF and USE attributes are very useful features for marking up large scenes and also many components can be reused, e.g. trees, buildings and houses in a city scene.

First, copy the node that corresponds to a DEF attribute and replace all nodes with that USE attribute name. This procedure is done when the XSLT processor is loading the X3D input document. After the node replacement, the XSLT processor traverses the tree structure and does the usual calculations.

6.2.2 User Interface

The JavaScript code in frameset handles button clicks. The buttons may be separated into two categories: movement and history buttons. These two types of buttons will be discussed in the next two sections.

6.2.2.1 Movement Buttons

The movement buttons consist of “move left”, “move right”, “move up”, “move down”, “move forward”, “move backward”, “rotate left”, “rotate right”, “rotate up”, and “rotate down” buttons. If any of these buttons is clicked, a Transform tag representing this movement is added as the parent node to the outer most Grouping or Shape node. For

example, the input document is initially loaded in a variable “xml” as shown in the following xml code:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="X3d2Vml.xsl"?>
<!DOCTYPE X3D SYSTEM " X3D2VML.dtd">
<X3D>
  <Scene>
    <Shape>
      <Appearance><Material/></Appearance>
      <Box size="200.0 300.0 400.0" />
    </Shape>
  </Scene>
</X3D>
```

After clicking “rotate left”, a “<Transform rotation=’ 0 1 0 0.314’ scale=’1 1 1’ translation=’0 0 0’></Transform>” tag is added to the xml tree structure. Variable xml becomes:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="X3d2Vml.xsl"?>
<!DOCTYPE X3D SYSTEM " X3D2VML.dtd">
<X3D>
  <Scene>
    <Transform rotation=’0 1 0 0.314’ scale=’1 1 1’ translation=’0 0 0’>
      <Shape>
        <Appearance><Material/></Appearance>
        <Box size="200.0 300.0 400.0" />
      </Shape>
    </Transform>
  </Scene>
</X3D>
```

This is not the end yet, the new result has to be output. This is done by the following line:

```
vmlFrame.document.write(xml.transformNode(xsl));
```

6.2.3.2 Back and Forward: History Buttons

The “Back” and “Forward” buttons allow back and forward movements. Nothing is done if there were no movements before. If the “Back” button is clicked, the outermost Transform node from the xml tree structure is removed and stored it in a temporary tree called “forwardNode”. The “forward” button is effective only if the last click is not one of the movement events and “forwardNode” is not NULL. The last movement is redone by removing the outermost Transform node from “forwardNode” and appending it to the xml tree as if a movement button is clicked.

In a sense, a Transform node is added for every movement button click, a Transform node is removed for every “back” button click, and a Transform node is added back for every “forward” button click.

6.2.2.2 User Interface Design

The users are able to change perspective easily with the user interface buttons. For this project, there are three user interface designs. They are shown from Figure 6-10 to Figure 6-12.

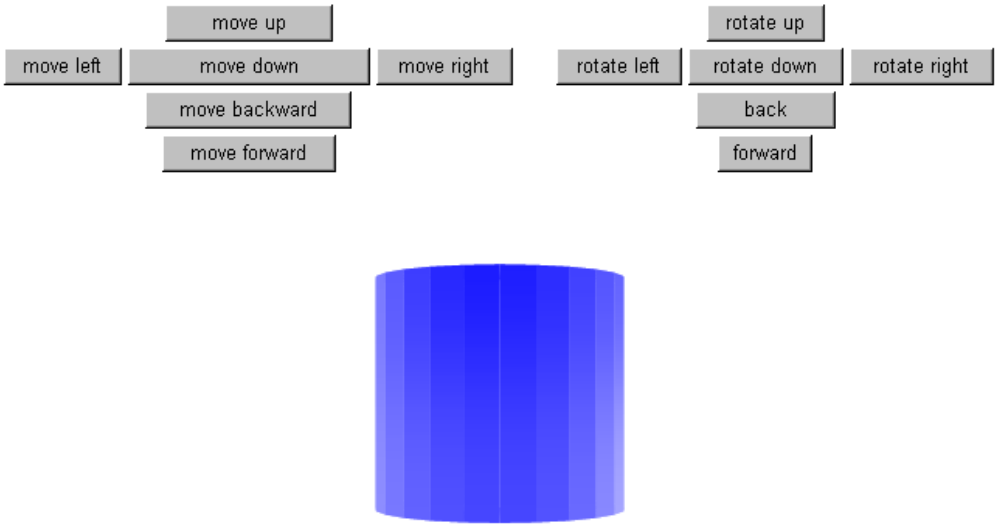


Figure 6-10. User Interface Design 1.

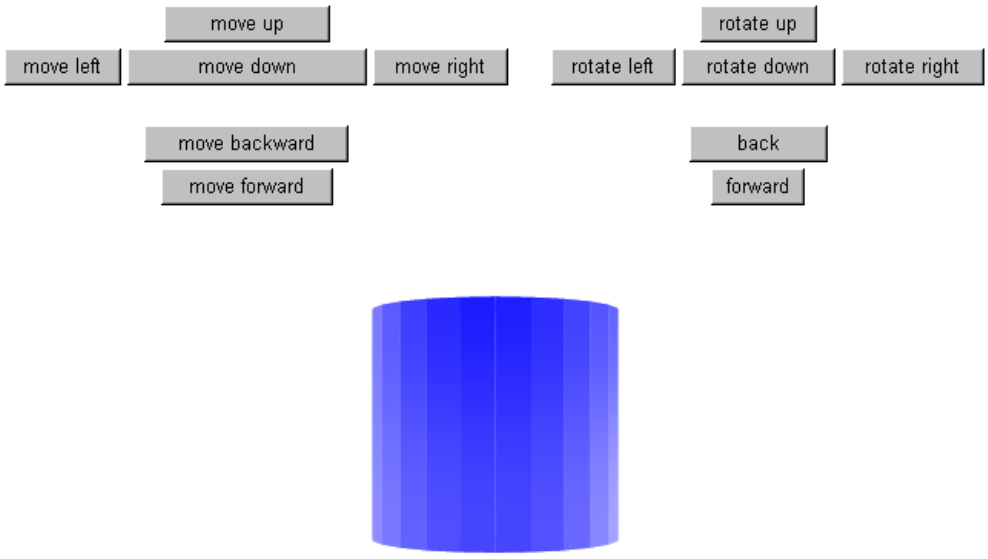


Figure 6-11. User Interface Design 2.

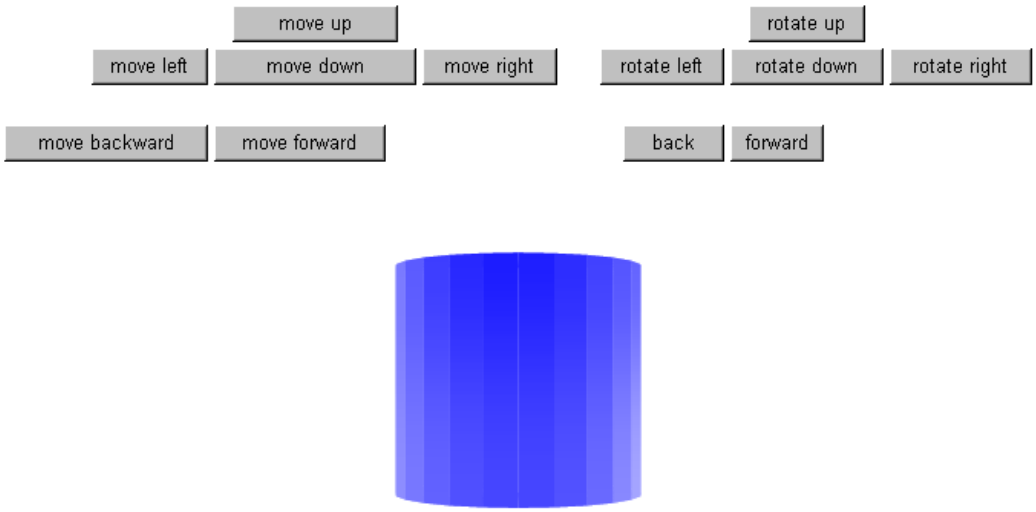


Figure 6-12. User Interface Design 3.

Three users were tested for each user interface design. The users were asked for general layout design. Then they were asked to perform some transformation tasks. Finally the users' overall feedback was obtained. A summary of their evaluation is organized in Table 6-2.

		User 1	**User 2**	**User 3**
Design 1	Design	Symetric both vertically and horizontally. Good.	Better than Design I, but "move backward", "move forward", "back" and "forward" button are confusing.	Better than Design 3, but "back" and "forward" buttons are confusing because they are not related to rotation buttons and they're grouped together.

	Usability	Have no problem to transform objects using three designs.	“move backward”, “move forward”, “back” and “forward” button are confusing at the beginning.	Have no problems to navigate.
	Overall Rating	Best of all three designs.	Average.	Better than Design 3, but little confusing.
Design 2	Design	Has less symmetric because there’s a line between top half and bottom half of the buttons	Clear and intuitive. The layout of the top half buttons are easy to understand because they are similar to the up, down, left, and right error keys on the key board.	Best design because there’s some space between rotation buttons and “back” and “forward” buttons.
	Usability	Have no problem to transform objects using three designs.	Have no problems to navigate.	Have no problems to navigate.
	Overall Rating	Average.	Best	Best
Design 3	Design	Top half of the buttons are symmetric, but the bottom half of the buttons look out of place.	Bottom half of the buttons look strange.	“move backward”, “move forward”, “back” and “forward” button are placed very unnaturally.

	Usability	Have no problem to transform objects using three designs.	Have no problems to navigate.	Have no problems to navigate.
	Overall Rating	Below average.	Average.	Below average

Table 6-2. User Interface Design Evaluation.

The user interface Design 2 is the best amongst the three designs in terms of design layout and usability. It is clear and intuitive. Hence Design 2 was used for the translator.

7. MAXIMUM LOAD

The maximum load was tested on a Windows 2000 PC with 2000 MHz CPU and 512 MB memory. The test result is organized in Table 7-1.

# of Polygons	Time for changing perspective (millisecond)							
	1st click	2nd click	3rd click	4th click	5th click	6th click	7th click	8th click
12	47	47	47	47	47	47	47	47
72	344	344	359	359	359	359	344	344
144	875	891	890	891	906	906	906	922
216	1609	1625	1640	1625	1656	1672	1656	1671
288	2688	2734	2765	2781	2781	2782	2796	2797

360	4079	4141	4172	4203	4219	4219	4235	4234
432	17859	16297	17609	18172	18828	18375	18656	17610
504	52703	55828	55297	54390	54812	56016	56375	56141
576	96156	97047	96453	96297	96356	96217	96305	96356

Table 7-1. Time for Changing Perspective for Each Click.

The box and cylinder were used for testing because they have a fixed number of polygons. The cone and sphere have more polygons for bigger objects. A box has 12 polygons and a cylinder has 72 polygons. The scene was changed eight times for each test. The processing time does NOT reduce from one click to the next because the whole xml tree structure is travered and all points are calculated for every click. The advantage of the calculation in the previous step could not be taken. The processing time is under one second when rendering 144 polygons. It becomes noticeable when rendering over 216 polygons. It takes about two seconds to render 288 polygons and 96 seconds to render 576 polygons. The performance of the translator is not good.

A Transform node was added into the xml tree structure for every movement, but it does not make much difference. It increases about 157 milliseconds for eight clicks for a scene that has 156 polygons.

8. LIMITATIONS

Since the core of the translator is XSLT and it supports back and forward history features, the whole xml tree structure needs to be traversed, generating points every time a button is clicked. The advantage of all the calculations we have done in the previous step could not be taken.

XSLT does not have the document.write() document object as in a DOM document. We could not directly output the VML tags from the JavaScript section of the X3dToVml.xsl file. What we did is to concatenate all information necessary into a very long string and

parse it later. This is a huge limitation of my translator, especially when things don't work.

9. CONCLUSION

A translator that transforms an X3D document to a VML document using XSLT was successfully implemented. The translator transforms the following primitive shapes: box, cylinder, cone, and sphere. It also allows color of a shape to be specified. It further has grouping and transformation features.

Object-oriented programming is used for our core stylesheet. It has classes Shape, Box, Cylinder, Cone, Sphere, Vector, Point, Intensity, RGB, TransformationMatrix, ProjectedPolygon, Group, and Transform.

The points representing primitive shapes are organized into rows and columns of a two dimensional array for each shape.

Transformations such as translation, rotation, and scaling are represented using 4X4 matrices. The product of the transformation matrix and a point gives the new location of the point.

The Phong lighting model is used to calculate the intensity of a point on a surface. It is chosen for providing a realistic effect. The amount of ambient, diffused, and specular light need to be specified for the calculation.

Gouraud shading is used to render polygon surfaces. The reason for choosing Gouraud shading is that it is fast, simple, and it also provides reasonably accurate surface shading. Gouroud shading renders polygons by linearly interpolating intensity values across the surface.

The <Group> and <Transform> tags are grouping tags which allow the manipulation of more than one object at a time. Translation, rotation, and scaling can be specified as attributes of the <Transform> tag.

The translator also has a user interface that allows users to transform scenes after they have been initially rendered. The user interface consists of “move left”, “move right”, “move up”, “move down”, “move forward”, “move backward”, “rotate left”, “rotate right”, “rotate up”, and “rotate down” buttons. The “back” and “forth” buttons allow users to going back and forth among movements.

The maximum number of polygons the translator could render is small. Clipping has not been implemented yet. The translator displays all polygons now. The shapes are generated and rendered even if they are background objects. The translator might render more polygons and run faster if only those polygons which are not covered by other polygons are displayed. Clipping is left for future work.

REFERENCES

[ANM97] VRML 2.0 Sourcebook. 2nd edition. A. Ames, D. Nadeau, J. Moreland. Wiley. 1997.

[ECMA99] Standard ECMA-262 ECMAScript Language Specification 3rd ed. http://www.ecma.ch/ecma1/stand/ecma-262.htm. ECMA. 1999.

[FD98] JavaScript: The Definitive Guide. 3rd edition. David Flanagan. O’Reilly & Associates. 1998.

[GR00] 3D Interactive VML. http://www.gersolutions.com/vml/. Gareth Richards. 2000.

[HB97] Computer Graphics C Version. 2nd edition. Donald Hearn, M. Pauline Baker. Prentic Hall. 1997.

[HD00] Beginning XML. David Hunter, with Curt Cagle, Dave Gibbons, Nikola Ozu, Jon Pinnock, Paul Spencer. Wrox Press Ltd. 2000

[HR01] Xml Bible. 2nd edition. Elliotte Rusty Harold. Hungry Minds, Inc. 2001.

[KM01] XSLT Programmer's Reference. 2nd edition. Michael Kay. Wrox Press Ltd. 2001.

[MK01] LiveGraphics3D Documentation. http://wwwvis.informatik.uni-stuttgart.de/~kraus/LiveGraphics3D/documentation.html. Martin Kraus. 2001.

[PS02] 3D Web Graphics without Plugins using SVG. Paungkaew Sangtrakulcharoen. http://www.cs.sjsu.edu/faculty/pollett/masters/.

[RE01] Learning XML. Erik T. Ray. 1st edition. O'Reilly & Associates. 2001.

[SRB02] 3D Computer Graphics A Mathematical Introduction with OpenGL. Samuel R. Buss. 2002.

[W00] 3D Computer Graphics. 3rd ed. Allan Watt. Pearson Education Limited. (Addison Wesley). 2000.

[W3C01] Scalable Vector Graphics (SVG) Specification 1.0. http://www.w3.org/TR/SVG/. W3C.

[W3C97] Extensible Markup Language (XML). http://www.w3.org/XML. W3C.

[W3C98] VML - the Vector Markup Language. http://www.w3.org/TR/NOTE-VML. W3C.

[W3C99] XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt. W3C.

[W3DC01] Extensible 3D (X3D) Graphics Working Group. http://www.web3d.org/x3d.html. Web 3D Consortium. 2001

APPENDIX A

This section lists all member variables and functions for the thirteen classes in the X3dToVml.xsl file.

Shape Class

Purpose: To project points in 3D to 2D using perspective projection, computes normal vector for each polygon, and returns shape polygons in 2D in a very long string.

DEFName defined name.
groupLevel

group level of this node relative to first Group or Transfrom node, starting from 0.
transformationMatrix transformation matrix that describes this node's position.
shapeArray a two dimensional array containing points represent this primitive.
normal1 this primitive's first normal (has total of two normals).
normal2 this primitive's second normal (has total of two normals).
rgb primitive's color, eg. blue, red, green.
Shape() default constructor.
compute() computes this shape.
computePointNormals() computes point normals (helper function for Shape.prototype.compute).
getShapePolygonsIn2D() gets polygon in 2D (helper function for Shape.prototype.compute).
toString() to string.
createShape() creates a shape.
beginShape() increments groups level when sees the beginning of a shape.
compareProjectedPolygons() comparing function for sorting.
endShape() do neccessary job when see the end of a shape.

Box Class

Purpose**:** To generate points for a box.

Box() constructs a box with given width, height, and depth.
createBox() creates a box with given width, height, and depth.

Cylinder Class

Purpose: To generate points for a cylinder.

Cylinder() constructs a cylinder with given height and radius.
CreateCylinder() creates a cylinder with given height and radius.

Cone Class

Purpose: To generate points for a cone.

Cone() constructs a cone with given height and bottom radius.

createCone() creates a cone with given height and bottom radius.

Sphere Class

Purpose: To generate points for a sphere.

Sphere() constructs a sphere with given radius.
CreateSphere() creates a sphere with given radius.

Vector Class

Purpose: To provide simple vector operation, such as normalization, dot product and compute magnitude.

VectorCount keep track of how many vector added so far.
VectorArray an array of three represents x, y, and z coordinates.
Vector() default constructor.
AddVector() adds a vector this vector.
AverageVector() averages this vector.
normalize() normalizes vector.
getX() gets x.
getY() gets y.
getZ() gets z.
setZ() sets z.
getMagnitude() gets magnitude.
dotProduct() compute a dot product of this vector and parameter vector.
toString() to string.

Point Class

Purpose: To perform transformation specified in transformationMatrix.

PointArray an array represents a points in 3D coordinates.
normal

this point’s normal.
Point() default constructor.
Compute() computes this point after a transformation.
GetPointXYCoordInString() gets x, y coordinate in string format.
getPointXCoord() gets x coordinate.
getPointYCoord() gets y coordinate.
getPointZCoord() gets z coordinate.
toString() to string.
createPoint() creates a point.

Intensity Class

Purpose: To calculate intensity using the Phong lighting model.

normal intensity normal.
IntensityValue intensity value.
Intensity() default constructor.
CalculateIntensity() calculates intensity given a normal vector.
setNormal() sets normal (helper function of calculateIntensity).
toString() to string.

ProjectedPolygon Class

Purpose: To sorts polygons from farthest to nearest.

DrawPolygonString a string has info for drawing a 2D polygon and its color, use its depth to arrange polygon from farthest to nearest.
depth this polygon’s depth.
ProjectedPolygon() constructs a ProjectedPolygon with a info string for drawing this polygon and its depth.
getDrawPolygonString() gets a info string for drawing this polygon.
getDepth() gets depth.
toString() to string.
createProjectedPolygon() creates a ProjectedPolygon with a info string for drawing this polygon

and its depth.

RGB Class

Purpose: To converts a color in rgb to hsv, sets saturation to intensity, then converts hsv back to rgb.

r floating point value represent red, from 0.0 to 1.0.
g floating point value represent green, from 0.0 to 1.0.
b floating point value represent blue, from 0.0 to 1.0.
RGB() constructs a RGB with r, g, b values.
rgbToHsvToRgb() converts rgb to hsv, sets saturation to intensity, then hsv to rgb.
getR() gets red (from 0 to 1).
getG() gets green (from 0 to 1).
getB() gets blue (from 0 to 1).
getRIn255Scale() gets red in 255 scale.
getGIn255Scale() gets green in 255 scale.
getBIn255Scale() gets blue in 255 scale.
toString() to string
createRGB() creats a RGB instance.

Group Class

Purpose: To group Shape, Group, or Transform nodes together.

beginGroup() creates a Group node.
endGroup() do neccessary job when sees the end of a Group tag.

Transform Class

Purpose: To support rotation, translation, and scaling.

createTransform() creates a Transform node.
endTransform() do neccessary job when sees the end of a Tranform tag.

TransformationMatrix Class

Purpose: To manipulate points in 3D.

TransformationMatrix() Creates an identity 4X4 matrix.
setIdentity() sets to an identity matrix.
MatrixMultiply() matrix multiplication (this = this * paramMatrix).
SetMatrixTranslation() sets translation.
SetMatrixScaling() Set scaling.
SetMatrixRotation() Sets rotation.
toString() returns a string representation

APPENDIX B

Source code for a sample X3D input file “threeBoxes.xml”.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="X3dToVml.xsl"?>

<!DOCTYPE X3D SYSTEM "X3DToVml.dtd">
<X3D>
    <Scene>
    <Transform rotation="0.57736 0.57736 -0.57736 2.09333333">

        <Transform translation="0 80 0">
            <Shape>
              <Appearance>
                    <Material diffuseColor="1 1 0"/>
            </Appearance>
                <Box size="30 160 30"/>
            </Shape>
        </Transform>

        <Transform translation="80 0 0"  rotation="0 0 1 1.57">
            <Shape>
              <Appearance>
                    <Material diffuseColor="1 0 0"/>
            </Appearance>
                <Box size="30 160 30"/>
            </Shape>
        </Transform>

        <Transform translation="0 0 -80" rotation="1 0 0 1.57">
            <Shape>
              <Appearance>
                <Material diffuseColor="0 0 1"/>
                 </Appearance>
                <Box size="30 160 30"/>
            </Shape>
        </Transform>

    </Transform>

    </Scene>
</X3D>
```

APPENDIX C

Source code for the “X3dToVml.dtd” file.

```
<!ENTITY % PrimitiveNodes "(Box | Cylinder | Cone | Sphere)">
<!ENTITY % GroupingNodes  "(Group | Transform)">
```

```
<!ENTITY % SceneNodes     "(%GroupingNodes;)*, (Shape)*">
<!ENTITY % ChildNodes  "(%SceneNodes;)*">

<!ELEMENT X3D (Scene)>
<!ELEMENT Scene
  (%SceneNodes;)>


<!ELEMENT Group %ChildNodes;>
<!ATTLIST Group
        DEF ID       #IMPLIED
        USE IDREF    #IMPLIED>


<!ELEMENT Transform %ChildNodes;>
<!ATTLIST Transform
  translation CDATA "0 0 0"
  scale       CDATA "1 1 1"
  rotation    CDATA "1 0 0 0">

<!ELEMENT Shape
  (Appearance?, (%PrimitiveNodes;)?)>

<!ELEMENT Appearance (Material)>

<!ELEMENT Material EMPTY>
<!ATTLIST Material
  diffuseColor     CDATA "0.0 0.0 1.0">

<!ELEMENT Box EMPTY>
<!ATTLIST Box
  size CDATA "50 50 50">

<!ELEMENT Cylinder EMPTY>
<!ATTLIST Cylinder
  height CDATA "100"
  radius CDATA "50">

<!ELEMENT Cone EMPTY>
<!ATTLIST Cone
  height CDATA "100"
  bottomRadius CDATA "50">

<!ELEMENT Sphere EMPTY>
<!ATTLIST Sphere
  radius CDATA "50">
```

APPENDIX D

Source code for the “X3dToVml.xsl” file.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:msxsl="urn:schemas-microsoft-com:xslt"
    xmlns:project="http://deliverable4">

<msxsl:script language="JavaScript1.2" implements-prefix="project">
<![CDATA[

// contains all elements for this scene, including Group, Transform and Shape
var sceneArray = new Array();

// contains all transformation matrix for Transform tags
var transformArray = new Array();
```

```
// used to determine which node is inside which node in sceneArray
var groupLevel = 0;

// tempority variable stores DEF name
var tempDEFName = "";

// will divide each circle into 12 pies, used by Cylinder, Cone, and Sphere
var angleConstant = 6;

// for "height >= 100", divide Cone and Sphere into layers
var bigHeightIncrement = 20;

// for "height < 100", divide Cone and Sphere into layers
var smallHeightIncrement = 5;

// contains projected array of 2D polygons use to compare depth
var projectedPolygonArray = new Array();

// default distance between shape and view point in z-axe
var viewDistance = 500;

// holds color for next shape to be created
var nextColor = null;

/*
   sets the global variable "nextColor" to this color
   @param r red compoment of the color to be set
   @param g green compoment of the color to be set
   @param b blue compoment of the color to be set
   PostCondition global variable "nextColor" is set to this color
*/
function setColor(r, g, b)
{

  var r1 = r - 0.0;
  var g1 = g - 0.0;
  var b1 = b - 0.0;
  nextColor = new RGB(r1, g1, b1);

  return nextColor.toString();
}


// BEGIN class TransformationMatrix
  /*
     default constructor
  */
    function TransformationMatrix()
  {
      var row0 = new Array(1, 0, 0, 0);

        var row1 = new Array(0, 1, 0, 0);

        var row2 = new Array(0, 0, 1, 0);

        var row3 = new Array(0, 0, 0, 1);
        this.matrix = new Array(row0, row1, row2, row3);
    }

    /*
        sets to an identity matrix
  */
    TransformationMatrix.prototype.setIdentity = function()
    {
      for (var row = 0; row < 4; ++row)
      {
        for (var column = 0; column < 4; ++column)
        {
          if (row == column)
          {
            this.matrix[row][column] = 1;
          } else
          {
            this.matrix[row][column] = 0;
          }
        }
```

```
    }
  }

  /*
        multiplies this matrix with the parameter matrix (this = this * paramMatrix)
        @param paramMatrix matrix to be multiplied with
  */
  TransformationMatrix.prototype.matrixMultiply = function(paramMatrix)
  {
    var temp = new TransformationMatrix();

    for (var row = 0; row < 4; ++row)
    {
      for (var column = 0; column < 4; ++column)
      {
        temp.matrix[row][column] = this.matrix[row][0]*paramMatrix.matrix[0][column] +
                                   this.matrix[row][1]*paramMatrix.matrix[1][column] +
                                   this.matrix[row][2]*paramMatrix.matrix[2][column] +
                                   this.matrix[row][3]*paramMatrix.matrix[3][column];
      }
    }

    for (var row = 0; row < 4; ++row)
    {
      for (var column = 0; column < 4; ++column)
      {
        this.matrix[row][column] = temp.matrix[row][column];
      }
    }
  }

  /*
        multiplies the parameter matrix with this matrix (this = paramMatrix * this)
        @param paramMatrix matrix to be multiplied with
  */
  TransformationMatrix.prototype.matrixPostMultiply = function(paramMatrix)
  {
    var temp = new TransformationMatrix();

    for (var row = 0; row < 4; ++row)
    {
      for (var column = 0; column < 4; ++column)
      {
        temp.matrix[row][column] = paramMatrix.matrix[row][0]*this.matrix[0][column] +
                                   paramMatrix.matrix[row][1]*this.matrix[1][column] +
                                   paramMatrix.matrix[row][2]*this.matrix[2][column] +
                                   paramMatrix.matrix[row][3]*this.matrix[3][column];
      }
    }

    for (var row = 0; row < 4; ++row)
    {
      for (var column = 0; column < 4; ++column)
      {
        this.matrix[row][column] = temp.matrix[row][column];
      }
    }
  }

  /*
     set this transformation matrix with a prespective projection looking from the
     view distance
  */
TransformationMatrix.prototype.addProjection = function()
  {
    this.matrix[3][2] = 1/viewDistance;
}

  /*
        undoes addProjection() function
  */
  TransformationMatrix.prototype.removeProjection = function()
  {
    this.matrix[3][2] = 0;
  }
```

```
  /*
     sets translation
     @param x moves x units along the x-axix
     @param y moves y units along the y-axix
     @param z moves z units along the z-axix
  */
  TransformationMatrix.prototype.setMatrixTranslation = function(x, y, z)
  {
    var temp = new TransformationMatrix();

    temp.setIdentity();
    temp.matrix[0][3] = x;
      temp.matrix[1][3] = y;
      temp.matrix[2][3] = z;

      this.matrixMultiply(temp);
  }

  /*
       sets scaling
       @param x scaling factor
       @param y scaling factor
       @param z scaling factor
  */
  TransformationMatrix.prototype.setMatrixScaling = function(x, y, z)
  {
    var temp = new TransformationMatrix();

    temp.setIdentity();
    temp.matrix[0][0] = x;
      temp.matrix[1][1] = y;
      temp.matrix[2][2] = z;

      this.matrixMultiply(temp);
  }

  /*
       calculates the matrix is rotated through an angle radianAngle around
      axis (x y z) vector
       @param x x compounent of the rotation axis
       @param y y compounent of the rotation axis
       @param z z compounent of the rotation axis
  */
  TransformationMatrix.prototype.setMatrixRotation = function(x, y, z, radianAngle)
  {
    var length = Math.sqrt(x*x + y*y + z*z);
    var cosA2 = Math.cos(radianAngle / 2);
    var sinA2 = Math.sin(radianAngle / 2);
    var a = sinA2 * x / length;
    var b = sinA2 * y / length;
    var c = sinA2 * z / length;
    var temp = new TransformationMatrix();

    temp.setIdentity();
    temp.matrix[0][0] = 1 - 2*b*b - 2*c*c;
      temp.matrix[0][1] = 2*a*b - 2*cosA2*c;
      temp.matrix[0][2] = 2*a*c + 2*cosA2*b;

      temp.matrix[1][0] = 2*a*b + 2*cosA2*c;
      temp.matrix[1][1] = 1 - 2*a*a - 2*c*c;
      temp.matrix[1][2] = 2*b*c - 2*cosA2*a;

      temp.matrix[2][0] = 2*a*c - 2*cosA2*b;
      temp.matrix[2][1] = 2*b*c + 2*cosA2*a;
      temp.matrix[2][2] = 1 - 2*a*a - 2*b*b;

      this.matrixMultiply(temp);
  }



/*
   to string
   @return string representation of the matrix
*/
  TransformationMatrix.prototype.toString = function()
```

```
    {
      var string = "Matrix {";

      for (var i = 0; i < 4; ++i)
        {
          string += "[ ";

          for (var j = 0; j < 4; ++j)
            {
              string += this.matrix[i][j] + " ";
            }

          string += "]";
        }

      string += " }";

      return string;
    }

// END class TransformationMatrix


// BEGIN class Vector
  /*
     default constructor
  */
  function Vector()
  {
    this.vectorCount = 0;
    this.vectorArray = new Array(0, 0, 0);
  }

  /*
     adds a vector this vector
     @param x x compoment of the vector
     @param y y compoment of the vector
     @param z z compoment of the vector
  */
  Vector.prototype.addVector = function(x, y, z)
  {
    this.vectorArray[0] += x;
    this.vectorArray[1] += y;
    this.vectorArray[2] += z;
    ++this.vectorCount;
  }

  /*
     averages the vector
  */
  Vector.prototype.averageVector = function()
  {
    this.vectorArray[0] /= this.vectorCount;
    this.vectorArray[1] /= this.vectorCount;
    this.vectorArray[2] /= this.vectorCount;

    this.vectorCount = -1;
  }

  /*
     normalizes the vector
  */
  Vector.prototype.normalize = function()
  {
    var magnitude = this.getMagnitude();

    this.vectorArray[0] /= magnitude;
    this.vectorArray[1] /= magnitude;
    this.vectorArray[2] /= magnitude;
  }

  /*
      gets x
      @return x value
  */
  Vector.prototype.getX = function()
```

```
{
  return this.vectorArray[0];
}

/*
     gets y
     @return y value
*/
Vector.prototype.getY = function()
  {
     return this.vectorArray[1];
}

  /*
       gets z
       @return z value
*/
Vector.prototype.getZ = function()
  {
     return this.vectorArray[2];
}

/*
   sets z
   @param z value to be set
*/
Vector.prototype.setZ = function(z)
{
  if (z != undefined)
  {
    this.vectorArray[2] = z;
  }
  }

/*
   gets magnitude
   @return magnitude value
*/
Vector.prototype.getMagnitude = function()
{
    var magnitude = this.vectorArray[0]*this.vectorArray[0] +
                    this.vectorArray[1]*this.vectorArray[1] +
                    this.vectorArray[2]*this.vectorArray[2];

  magnitude = Math.sqrt(magnitude);

  return magnitude;
}

/*
   computes a dot product of this vector and paramter vector
   @param vector to be computed with
   @return dot product value
*/
Vector.prototype.dotProduct = function(vector)
{
  if (vector != undefined)
  {
    return this.vectorArray[0]*vector.vectorArray[0] +
           this.vectorArray[1]*vector.vectorArray[1] +
           this.vectorArray[2]*vector.vectorArray[2];
  }
  else
  {
    return "Vector.prototype.dotProduct's parameter is undefined !";
  }
}

/*
   to string
   @return string representation of the vector
*/
Vector.prototype.toString = function()
{
  var string = "Vector, vectorCount: " + this.vectorCount +
               "(" +
```

```
                this.vectorArray[0] + ", " +
                this.vectorArray[1] + ", " +
                this.vectorArray[2] +
                ")";

    return string;
  }

// END class Vector

// BEGIN class Point
  /*
     default constructor
     @param x value of the point
     @param y value of the point
     @param z value of the point
  */
  function Point(x, y, z)
  {
    this.pointArray = new Array(x, y, z, 1);
    this.normal = new Vector();
  }

  /*
     computes this point after a transformation
     @param transformationMatrix stores info the point will be transformed
  */
  Point.prototype.compute = function(transformationMatrix)
  {
    var tempArray = new Array(0, 0, 0, 1);

    for (var i = 0; i < 4; ++i)
    {
      var result = 0;

      for (var j = 0; j < 4; ++j)
      {
            result += transformationMatrix.matrix[i][j] * this.pointArray[j];
      }

      tempArray[i] = result;
    }

    for (i = 0; i < 3; ++i)
    {
      tempArray[i] /= tempArray[3];
    }


    for (i = 0; i < 4; ++i)
    {
      this.pointArray[i] = tempArray[i];
    }

  }

  /*
     gets x, y coordinate in string format
     @return x, y coordinate in string format
  */
  Point.prototype.getPointXYCoordInString = function()
  {
    var tempPoint = new Point(this.pointArray[0], this.pointArray[1],
this.pointArray[2]);
    var tempMatrix = new TransformationMatrix();
    tempMatrix.addProjection();

    tempPoint.compute(tempMatrix);

    var string = tempPoint.pointArray[0] + " " + tempPoint.pointArray[1] + " ";

    return string;
  }

  /*
     get x coordinate value
```

```
    return x coordinate value
  */
  Point.prototype.getPointXCoord = function()
  {
    return this.pointArray[0];
  }

  /*
       get y coordinate value
       return x coordinate value
  */
  Point.prototype.getPointYCoord = function()
  {
    return this.pointArray[1];
  }

  /*
         get z coordinate value
         return z coordinate value
  */
  Point.prototype.getPointZCoord = function()
  {
    return this.pointArray[2];
  }

  /*
     to string
     @return string representation of the point
  */
  Point.prototype.toString = function()
  {
    var string = "Point[" +
                 this.pointArray[0] + " " +
                 this.pointArray[1] + " " +
                 this.pointArray[2] + " " +
                 this.pointArray[3] +
                 "]" +
                 this.normal.toString();

    return string;
  }

  /*
     creates a point with the specified x, y, z coordinates
     @param x coordinate
     @param y coordinate
     @param z coordinate
  */
  function createPoint(x, y, z)
  {
    var point = new Point(x, y, z);
    var transformationMatrix = new TransformationMatrix();

    return point.toString();
  }
// END class Point

// BEGIN class Intensity
  // used to calculate intensity
  var lightSourceVector = new Vector();

  // light source vector
  lightSourceVector.setZ(-1);

  // viewing vector
  var viewingVector = new Vector();

  viewingVector.setZ(-1);

  var halfwayVector = new Vector();

  halfwayVector.addVector(lightSourceVector.getX(),
                          lightSourceVector.getY(),
                          lightSourceVector.getZ()
                         );
```

```
  halfwayVector.addVector(viewingVector.getX(),
                          viewingVector.getY(),
                          viewingVector.getZ()
                         );

  halfwayVector.normalize();

  /*
     default constructor
  */
  function Intensity()
    {
      this.normal = new Vector();
      this.intensityValue = 0;
    }

    // (Pa) Phuong ambient reflectivity coefficient
    Intensity.ambient = 0.4;

  // (Pd) Phuong diffuse reflectivity coefficient
    Intensity.diffuse = 0.3;

    // (Ps) Phuong specular reflectivity coefficient
    Intensity.specular = 0.3;

    /*
       calculates intensity given a normal vector
       @param normal calculates intensity given a normal vector
    */
    Intensity.prototype.calculateIntensity = function(normal)
    {
      this.setNormal(normal);
      this.normal.normalize();

      this.intensityValue = Intensity.ambient +
                            Intensity.diffuse*lightSourceVector.dotProduct(this.normal) +
                            Intensity.specular*halfwayVector.dotProduct(this.normal)*
                            halfwayVector.dotProduct(this.normal);

      return this.intensityValue;
    }

    /*
       sets normal (helper function of calculateIntensity)
       @param normal to be set
    */
    Intensity.prototype.setNormal = function(normal)
    {
          if (normal != undefined)
          {
            this.normal.addVector(normal.getX(),
                                  normal.getY(),
                                  normal.getZ()
                                 );
          }
    }

    /*
       to string
       @return string representation of the Intensity
    */
    Intensity.prototype.toString = function()
    {
      var string = "Intensity, intensityValue:" +
                   this.intensityValue +
                   this.normal.toString();

      return string;
    }
// END class Intensity

// BEGIN class ProjectedPolygon
  /*
     constructs a ProjectedPolygon with a info string for drawing this polygon and
     its depth
     @param drawPolygonString string contains info about the polygon
```

```
    @param depth represents the polygon
  */
  function ProjectedPolygon(drawPolygonString, depth)
  {
    this.drawPolygonString = drawPolygonString;
    this.depth = depth;
  }

  /*
     gets a info string for drawing this polygon
     @return string for drawing this polygon
  */
  ProjectedPolygon.prototype.getDrawPolygonString = function()
  {
    return this.drawPolygonString;
  }

  /*
     gets depth
     @return depth
  */
  ProjectedPolygon.prototype.getDepth = function()
    {
      return this.depth;
  }

  /*
     to string
     @return string representation of the ProjectedPolygon
  */
  ProjectedPolygon.prototype.toString = function()
  {
    var string = "ProjectedPolygon, drawPolygonString: " +
                 this.drawPolygonString +
                 " depth: " + this.depth;
       return string;
  }

  /*
     creates a ProjectedPolygon with a info string for drawing this polygon and
     its depth.
     The newly created ProjectedPolygon is pushed into the projectedPolygonArray.

  */
  function createProjectedPolygon(drawPolygonString, depth)
  {
    var projectedPolygon = new ProjectedPolygon(drawPolygonString, depth);

    projectedPolygonArray.push(projectedPolygon);

    var string = projectedPolygon.toString() +
                 "projectedPolygonArray.length: " + projectedPolygonArray.length;

    return string;
  }

// END class ProjectedPolygon

// BEGIN class Shape
  /*
    default constructor
  */
  function Shape()
  {
    this.DEFName = "";
    this.type = "";

    this.shapeArray = new Array();
    this.normal1 = new Vector();
    this.normal2 = new Vector();
    this.rgb = new RGB(0, 0, 1);

    this.transformationMatrix = new TransformationMatrix();

  }
```

```
/*
   computes this shape
   @param transformationMatrix position the shape will be transformed to
*/
Shape.prototype.compute = function(transformationMatrix)
{

  if (transformationMatrix != undefined)
  {
    this.transformationMatrix.matrixMultiply(transformationMatrix);
  }


  for (var i = 0; i < this.shapeArray.length; ++i)
  {
    for (var j = 0; j < this.shapeArray[i].length; ++j)
    {
      this.shapeArray[i][j].compute(this.transformationMatrix);
    }
  }

  this.computePointNormals();

  return this.getShapePolygonsIn2D();
}


/*
   computes point normals (helper function for Shape.prototype.compute)
*/
Shape.prototype.computePointNormals = function()
{
  var string = "";

  var x1;
  var y1;
  var z1;

  var x2;
  var y2;
  var z2;

  var x3;
  var y3;
  var z3;

  var A;
  var B;
  var C;

  for (var i = this.shapeArray.length - 1; i >= 1; --i)
  {
    for (var j = this.shapeArray[i].length - 1; j >= 1; --j)
    {
       x1 = this.shapeArray[i][j].getPointXCoord();
         y1 = this.shapeArray[i][j].getPointYCoord();
       z1 = this.shapeArray[i][j].getPointZCoord();

       x2 = this.shapeArray[i][j-1].getPointXCoord();
       y2 = this.shapeArray[i][j-1].getPointYCoord();
       z2 = this.shapeArray[i][j-1].getPointZCoord();

       x3 = this.shapeArray[i-1][j-1].getPointXCoord();
       y3 = this.shapeArray[i-1][j-1].getPointYCoord();
       z3 = this.shapeArray[i-1][j-1].getPointZCoord();

       A = y1*(z2 - z3) + y2*(z3 - z1) + y3*(z1 - z2);
       B = z1*(x2 - x3) + z2*(x3 - x1) + z3*(x1 - x2);
       C = x1*(y2 - y3) + x2*(y3 - y1) + x3*(y1 - y2);

       //record normals into these four points
       this.shapeArray[i][j].normal.addVector(A, B, C);
       this.shapeArray[i][j-1].normal.addVector(A, B, C);
       this.shapeArray[i-1][j-1].normal.addVector(A, B, C);
       this.shapeArray[i-1][j].normal.addVector(A, B, C);
    }
```

```
    // calculate normals of a polygon forming by first and last 4 points
    x1 = this.shapeArray[i][0].getPointXCoord();
    y1 = this.shapeArray[i][0].getPointYCoord();
    z1 = this.shapeArray[i][0].getPointZCoord();

    x2 = this.shapeArray[i][this.shapeArray[i].length-1].getPointXCoord();
    y2 = this.shapeArray[i][this.shapeArray[i].length-1].getPointYCoord();
    z2 = this.shapeArray[i][this.shapeArray[i].length-1].getPointZCoord();


    x3 = this.shapeArray[i-1][this.shapeArray[i].length-1].getPointXCoord();
    y3 = this.shapeArray[i-1][this.shapeArray[i].length-1].getPointYCoord();
    z3 = this.shapeArray[i-1][this.shapeArray[i].length-1].getPointZCoord();

    A = y1*(z2 - z3) + y2*(z3 - z1) + y3*(z1 - z2);
    B = z1*(x2 - x3) + z2*(x3 - x1) + z3*(x1 - x2);
    C = x1*(y2 - y3) + x2*(y3 - y1) + x3*(y1 - y2);

   //record normals into these four points
   this.shapeArray[i][0].normal.addVector(A, B, C);
   this.shapeArray[i][this.shapeArray[i].length-1].normal.addVector(A, B, C);
   this.shapeArray[i-1][this.shapeArray[i].length-1].normal.addVector(A, B, C);
   this.shapeArray[i-1][0].normal.addVector(A, B, C);


  }

  return string;
}

 /*
    gets polygon in 2D (helper function for Shape.prototype.compute)
    @return string of the polygon in 2D
 */
 Shape.prototype.getShapePolygonsIn2D = function()
   {
      var string = "";
      var intensity1 = new Intensity();
      var intensity2 = new Intensity();
      var I1 = 0;
      var I2 = 0;
      var rgbTemp = new RGB(this.rgb.getR(), this.rgb.getG(), this.rgb.getB());

      var z1 = 0;
      var z2 = 0;
      var z3 = 0;
      var z4 = 0;
      var minDepth = 0;

      var maxDepth = 0;

      var depth = 0;

      var i = this.shapeArray.length - 1;
      var endIndex = 1;


      for (var i = this.shapeArray.length - 1; i >= 1; --i)
      {
        for (var j = this.shapeArray[i].length - 1; j >= 1; --j)
        {
          string = "";

          if ((this.type == "box") &&
              ((i == this.shapeArray.length - 1) || (i == 1)))
          {
            string += this.shapeArray[i][j].getPointXYCoordInString() + " ";
            string += this.shapeArray[i][j-1].getPointXYCoordInString() + " ";
            string += this.shapeArray[i-1][j-1].getPointXYCoordInString() + " ";
            string += this.shapeArray[i-1][j].getPointXYCoordInString() + " ";

            var rgbTemp = new RGB(this.rgb.getR(), this.rgb.getG(), this.rgb.getB());

                   string += rgbTemp.getRIn255Scale() + " ";
```

```
        string += rgbTemp.getGIn255Scale() + " ";
  string += rgbTemp.getBIn255Scale() + " ";

  string += rgbTemp.getRIn255Scale() + " ";
        string += rgbTemp.getGIn255Scale() + " ";
  string += rgbTemp.getBIn255Scale() + " ";
}
else
{
this.shapeArray[i][j].normal.averageVector();
this.shapeArray[i][j-1].normal.averageVector();
this.shapeArray[i-1][j-1].normal.averageVector();
this.shapeArray[i-1][j].normal.averageVector();

this.shapeArray[i][j].normal.normalize();
this.shapeArray[i][j-1].normal.normalize();
this.shapeArray[i-1][j-1].normal.normalize();
this.shapeArray[i-1][j].normal.normalize();


this.normal1.addVector(this.shapeArray[i][j].normal.getX(),
                       this.shapeArray[i][j].normal.getY(),
                       this.shapeArray[i][j].normal.getZ()
                      );

this.normal1.addVector(this.shapeArray[i][j-1].normal.getX(),
                       this.shapeArray[i][j-1].normal.getY(),
                       this.shapeArray[i][j-1].normal.getZ()
                      );

this.normal2.addVector(this.shapeArray[i-1][j].normal.getX(),
                       this.shapeArray[i-1][j].normal.getY(),
                       this.shapeArray[i-1][j].normal.getZ()
                      );

this.normal2.addVector(this.shapeArray[i-1][j-1].normal.getX(),
                        this.shapeArray[i-1][j-1].normal.getY(),
                        this.shapeArray[i-1][j-1].normal.getZ()
                      );


this.normal1.averageVector();
this.normal2.averageVector();

this.normal1.normalize();
this.normal2.normalize();


string += this.shapeArray[i][j].getPointXYCoordInString() + " ";
string += this.shapeArray[i][j-1].getPointXYCoordInString() + " ";
string += this.shapeArray[i-1][j-1].getPointXYCoordInString() + " ";
string += this.shapeArray[i-1][j].getPointXYCoordInString() + " ";

I1 = intensity1.calculateIntensity(this.normal1);
I2 = intensity2.calculateIntensity(this.normal2);

rgbTemp = new RGB(this.rgb.getR(), this.rgb.getG(), this.rgb.getB());
rgbTemp.rgbToHsvToRgb(I1);
string += rgbTemp.getRIn255Scale() + " ";
string += rgbTemp.getGIn255Scale() + " ";
string += rgbTemp.getBIn255Scale() + " ";


rgbTemp = new RGB(this.rgb.getR(), this.rgb.getG(), this.rgb.getB());
       rgbTemp.rgbToHsvToRgb(I2);
       string += rgbTemp.getRIn255Scale() + " ";
       string += rgbTemp.getGIn255Scale() + " ";
string += rgbTemp.getBIn255Scale() + " ";
}
// find depth for this polygon (average min and max depth)
z1 = this.shapeArray[i][j].getPointZCoord();
z2 = this.shapeArray[i][j-1].getPointZCoord();
z3 = this.shapeArray[i-1][j-1].getPointZCoord();
z4 = this.shapeArray[i-1][j].getPointZCoord();

minDepth = Math.min(z1, Math.min(z2, Math.min(z3, z4)));
```

```
            maxDepth = Math.max(z1, Math.max(z2, Math.max(z3, z4)));

            depth = (minDepth + maxDepth) / 2;
            createProjectedPolygon(string, depth);

          }


          // take care of a polygon formed by first and last 4 points
          string = "";

          if ((this.type == "box") &&
                    ((i == this.shapeArray.length - 1) || (i == 1)))
          {
          string += this.shapeArray[i][0].getPointXYCoordInString() + " ";
            string += this.shapeArray[i][this.shapeArray[i].length-
1].getPointXYCoordInString() + " ";
            string += this.shapeArray[i-1][this.shapeArray[i].length-
1].getPointXYCoordInString() + " ";
            string += this.shapeArray[i-1][0].getPointXYCoordInString() + " ";

                var rgbTemp = new RGB(this.rgb.getR(), this.rgb.getG(),
                                      this.rgb.getB());

                string += rgbTemp.getRIn255Scale() + " ";
                string += rgbTemp.getGIn255Scale() + " ";
                string += rgbTemp.getBIn255Scale() + " ";

                string += rgbTemp.getRIn255Scale() + " ";
                  string += rgbTemp.getGIn255Scale() + " ";
                  string += rgbTemp.getBIn255Scale() + " ";

          }
          else
          {
          this.shapeArray[i][0].normal.averageVector();
          this.shapeArray[i][this.shapeArray[i].length-1].normal.averageVector();
          this.shapeArray[i-1][this.shapeArray[i].length-1].normal.averageVector();
          this.shapeArray[i-1][0].normal.averageVector();

          this.shapeArray[i][0].normal.normalize();
          this.shapeArray[i][this.shapeArray[i].length-1].normal.normalize();
          this.shapeArray[i-1][this.shapeArray[i].length-1].normal.normalize();
          this.shapeArray[i-1][0].normal.normalize();


          this.normal1.addVector(this.shapeArray[i][0].normal.getX(),
                                 this.shapeArray[i][0].normal.getY(),
                                 this.shapeArray[i][0].normal.getZ()
                               );
          this.normal1.addVector(
            this.shapeArray[i][this.shapeArray[i].length-1].normal.getX(),
            this.shapeArray[i][this.shapeArray[i].length-1].normal.getY(),
            this.shapeArray[i][this.shapeArray[i].length-1].normal.getZ()
          );


          this.normal2.addVector(this.shapeArray[i-1][0].normal.getX(),
                                 this.shapeArray[i-1][0].normal.getY(),
                                 this.shapeArray[i-1][0].normal.getZ()
          );


          this.normal2.addVector(
           this.shapeArray[i-1][this.shapeArray[i].length-1].normal.getX(),
            this.shapeArray[i-1][this.shapeArray[i].length-1].normal.getY(),
            this.shapeArray[i-1][this.shapeArray[i].length-1].normal.getZ()
         );


          this.normal1.averageVector();
          this.normal2.averageVector();

          this.normal1.normalize();
          this.normal2.normalize();
```

```
          string += this.shapeArray[i][0].getPointXYCoordInString() + " ";
          string += this.shapeArray[i][this.shapeArray[i].length-
1].getPointXYCoordInString() + " ";
          string += this.shapeArray[i-1][this.shapeArray[i].length-
1].getPointXYCoordInString() + " ";
          string += this.shapeArray[i-1][0].getPointXYCoordInString() + " ";

          I1 = intensity1.calculateIntensity(this.normal1);
          I2 = intensity2.calculateIntensity(this.normal2);

          rgbTemp = new RGB(this.rgb.getR(), this.rgb.getG(), this.rgb.getB());
          rgbTemp.rgbToHsvToRgb(I1);
          string += rgbTemp.getRIn255Scale() + " ";
          string += rgbTemp.getGIn255Scale() + " ";
          string += rgbTemp.getBIn255Scale() + " ";


          rgbTemp = new RGB(this.rgb.getR(), this.rgb.getG(), this.rgb.getB());
          rgbTemp.rgbToHsvToRgb(I2);
          string += rgbTemp.getRIn255Scale() + " ";
          string += rgbTemp.getGIn255Scale() + " ";
          string += rgbTemp.getBIn255Scale() + " ";
      }

          // find depth for this polygon (average min and max depth)
          z1 = this.shapeArray[i][0].getPointZCoord();
          z2 = this.shapeArray[i][this.shapeArray[i].length-1].getPointZCoord();
          z3 = this.shapeArray[i-1][this.shapeArray[i].length-1].getPointZCoord();
          z4 = this.shapeArray[i-1][0].getPointZCoord();

          minDepth = Math.min(z1, Math.min(z2, Math.min(z3, z4)));
          maxDepth = Math.max(z1, Math.max(z2, Math.max(z3, z4)));

          depth = (minDepth + maxDepth) / 2;
          createProjectedPolygon(string, depth);

        }

        return string;
  }

  /*
     to string
     @return string representation of the shape
  */
  Shape.prototype.toString = function()
  {
    var string = "Shape{" +
                 "sceneArray length: " + sceneArray.length;

    string += " shape transformationMatrix: " + this.transformationMatrix.toString();
    string += " normal1:" + this.normal1.toString();
    string += " normal2:" + this.normal2.toString();

    for (var i = 0; i < this.shapeArray.length; ++i)
    {
      string += "shape[" + i + "] ";

      for (var j = 0; j < this.shapeArray[i].length; ++j)
      {
        string += this.shapeArray[i][j].toString();
      }
    }

    string += "}";

    return string;
  }

  /*
     creates a shape
  */
  function createShape()
  {
    var shape = new Shape();
```

```
    return shape.toString();

  }

  /*
     comparing function for sorting projected polygons
  */
  function compareProjectedPolygons(a, b)
  {
    if (a.getDepth() < b.getDepth())
    {
      return 1;
    }
    else if (a.getDepth() > b.getDepth())
        {
              return -1;
    }
    else
    {
      return 0;
    }
  }

  /*
     do neccessary job when see the end of a shape
  */
  function endShape()
  {
     if (nextColor != null)
     {
       sceneArray[sceneArray.length -1].rgb = nextColor;
     }


     sceneArray[sceneArray.length -1].compute();

     return "";

  }
// END class Shape

/*
   computes each shapes position
   @return string of polygons in 2D
*/
function endScene()
{
    var string = "";
    string += "projectedPolygonArray.length: " + projectedPolygonArray.length;
    string ="";

    projectedPolygonArray.sort(compareProjectedPolygons);

    for (var i = 0; i < projectedPolygonArray.length; ++i)
      {
            string += projectedPolygonArray[i].getDrawPolygonString();
    }

    return string;

}


// BEGIN class Box
  /*
     constructs a box with given width, height, and depth
     @param width width of the box
     @param height height of the box
     @param depth depth of the box
  */
  function Box(width, height, depth)
  {
    this.shapeArray = new Array();


    this.type= "box";
```

```
  this.transformationMatrix = new TransformationMatrix();


  for (var i = transformArray.length - 1; i >= 0; --i)
      {
        this.transformationMatrix.matrixPostMultiply(transformArray[i]);
  }


  var x = width / 2;
  var y = height / 2;
  var z = depth / 2;

  var point0T = new Point(0, y, 0);
      var point1T = new Point(0, y, 0);
      var point2T = new Point(0, y, 0);
      var point3T = new Point(0, y, 0);
      var boxArrayTop = new Array();
      boxArrayTop.push(point0T);
      boxArrayTop.push(point1T);
      boxArrayTop.push(point2T);
      boxArrayTop.push(point3T);
  this.shapeArray.push(boxArrayTop);

  var point0 = new Point(x, y, z);
  var point1 = new Point(-x, y, z);
  var point2 = new Point(-x, y, -z);
  var point3 = new Point(x, y, -z);
  var boxArray0 = new Array();
  boxArray0.push(point0);
  boxArray0.push(point1);
  boxArray0.push(point2);
  boxArray0.push(point3);
  //boxArray0.push(point0);
  this.shapeArray.push(boxArray0);

  var point4 = new Point(x, -y, z);
  var point5 = new Point(-x, -y, z);
  var point6 = new Point(-x, -y, -z);
  var point7 = new Point(x, -y, -z);
  var boxArray1 = new Array();
  boxArray1.push(point4);
  boxArray1.push(point5);
  boxArray1.push(point6);
  boxArray1.push(point7);
  //boxArray1.push(point4);
  this.shapeArray.push(boxArray1);

  var point0B = new Point(0, -y, 0);
    var point1B = new Point(0, -y, 0);
      var point2B = new Point(0, -y, 0);
      var point3B = new Point(0, -y, 0);
      var boxArrayBottom = new Array();
      boxArrayBottom.push(point0B);
      boxArrayBottom.push(point1B);
      boxArrayBottom.push(point2B);
      boxArrayBottom.push(point3B);
  this.shapeArray.push(boxArrayBottom);

  sceneArray.push(this);
}

/*
   instance of Box prototype inherites from Shape's prototype
*/
Box.prototype = new Shape();

/*
   creates a box with given width, height, and depth
   @param width width of the box
       @param height height of the box
   @param depth depth of the box
*/
function createBox(width, height, depth)
{
```

```
    var box = new Box(width, height, depth);

    return box.transformationMatrix.toString();


  }
// END class Box

// BEGIN class Cylinder
  /*
     constructs a cylinder with given height and radius
     @param height height of the cylinder
     @param radius of the cylinder
  */
  function Cylinder(height, radius)
  {

    this.shapeArray = new Array();

    this.type= "cylinder";

    this.transformationMatrix = new TransformationMatrix();

    for (var i = transformArray.length - 1; i >= 0; --i)
        {
          this.transformationMatrix.matrixPostMultiply(transformArray[i]);
    }

    var y = height / 2;
    //var angle = angleConstant / radius;
    var angle = Math.PI / 12;
    var centerRadius = 0.05;

    var cylinderArrayTop = new Array();

    for (var i = 0; i < Math.PI * 2; i += angle)
        {
          cylinderArrayTop.push(new Point(centerRadius * Math.cos(i), y,
                                          centerRadius * Math.sin(i)));
    }

    this.shapeArray.push(cylinderArrayTop);

    var cylinderArray0 = new Array();

    for (var i = 0; i < Math.PI * 2; i += angle)
    {
      cylinderArray0.push(new Point(radius * Math.cos(i), y, radius * Math.sin(i)));
    }

    this.shapeArray.push(cylinderArray0);

    var cylinderArray1 = new Array();

    for (var i = 0; i < Math.PI * 2; i += angle)
    {
      cylinderArray1.push(new Point(radius * Math.cos(i), -y, radius * Math.sin(i)));
    }

    this.shapeArray.push(cylinderArray1);

    var cylinderArrayBottom = new Array();

        for (var i = 0; i < Math.PI * 2; i += angle)
        {
          cylinderArrayBottom.push(new Point(centerRadius * Math.cos(i), -y,
                                             centerRadius * Math.sin(i)));
        }

    this.shapeArray.push(cylinderArrayBottom);

    sceneArray.push(this);
  }

  /*
     instance of Cylinder prototype inherites from Shape's prototype
```

```
  */
  Cylinder.prototype = new Shape();

  /*
     creates a cylinder with given height and radius
     @param height height of the cylinder
     @param radius of the cylinder
  */
  function createCylinder(height, radius)
  {
    var cylinder = new Cylinder(height, radius);

    return cylinder.toString();
  }

// END class Cylinder

// BEGIN class Cone
  /*
     constructs a cone with given height and bottom radius
     @param height height of the cone
     @param bottomRadius bottom radius of the cone
  */
  function Cone(height, bottomRadius)
  {

    this.shapeArray = new Array();

    this.type= "Cone";

    this.transformationMatrix = new TransformationMatrix();

    for (var i = transformArray.length - 1; i >= 0; --i)
        {
          this.transformationMatrix.matrixPostMultiply(transformArray[i]);
    }

    var minRealHeight = -height / 2;
    var proportionHeight = height;
    var heightIncrement = (height >= 100) ? bigHeightIncrement : smallHeightIncrement;

    var radius = bottomRadius;
    var angle = Math.PI / 12;
    var centerRadius = 0.05;
    var coneArray = new Array();

    for (var i = 0; i < Math.PI * 2; i += angle)
        {
          coneArray.push(new Point(centerRadius * Math.cos(i), minRealHeight,
                         centerRadius * Math.sin(i)));
        }

    this.shapeArray.push(coneArray);

    for (; minRealHeight <= height / 2;
         minRealHeight += heightIncrement, proportionHeight -= heightIncrement)
    {
      radius = bottomRadius * proportionHeight / height;
      coneArray = new Array();

      for (var i = 0; i < Math.PI * 2; i += angle)
      {
        coneArray.push(new Point(radius * Math.cos(i), minRealHeight,
                                 radius * Math.sin(i)));
      }

      this.shapeArray.push(coneArray);

    }

    if (minRealHeight != height / 2)
    {
      coneArray = new Array();
      radius = 1;
      minRealHeight = height / 2;
```

```
      for (var i = 0; i < Math.PI * 2; i += angle)
      {
        coneArray.push(new Point(radius * Math.cos(i), minRealHeight,
                                 radius * Math.sin(i)));
      }

      this.shapeArray.push(coneArray);
    }

    sceneArray.push(this);
  }


  /*
     instance of Cone prototype inherites from Shape's prototype
  */
  Cone.prototype = new Shape();

  /*
     creates a cone with given height and bottom radius
     @param height height of the cone
     @param bottomRadius bottom radius of the cone
  */
  function createCone(height, bottomRadius)
  {
    var cone = new Cone(height, bottomRadius);

    return cone.toString();
  }

// END class Cone

// BEGIN class Sphere
  /*
     constructs a sphere with given radius
     @param radius of the sphere
  */
  function Sphere(radius)
  {
    this.shapeArray = new Array();

    this.type= "sphere";

    this.transformationMatrix = new TransformationMatrix();

    for (var i = transformArray.length - 1; i >= 0; --i)
        {
          this.transformationMatrix.matrixPostMultiply(transformArray[i]);
    }

    var heightIncrement = (radius * 2 >= 100) ?
                          bigHeightIncrement :
                          smallHeightIncrement;
    var angle = Math.PI / 12;
    var sphereArray = new Array();
    var r = 0;

    for (var height = -radius; height <= radius; height += heightIncrement)
    {
      sphereArray = new Array();
      r = Math.cos(Math.asin(Math.abs(height) / radius)) * radius;

      if (r == 0)
      {
        r = 1;
      }

      for (var i = 0; i < Math.PI * 2; i += angle)
      {
        sphereArray.push(new Point(r * Math.cos(i), height, r * Math.sin(i)));
      }

      this.shapeArray.push(sphereArray);
    }
```

```
    sceneArray.push(this);
  }

  /*
    instance of Sphere prototype inherites from Shape's prototype
  */
  Sphere.prototype = new Shape();

  /*
     creates a sphere with given radius
     @param radius radius of the sphere
  */
  function createSphere(radius)
  {
    var sphere = new Sphere(radius);

    return sphere.toString();
  }

// END class Sphere

// BEGIN class RGB
  /*
     constructs a RGB with the specified r, g, b components
     @param r red component in floating point ranging from 0, to 1
     @param g green component in floating point ranging from 0, to 1
     @param b blue component in floating point ranging from 0, to 1component
  */
  function RGB(r, g, b)
  {
    if ((r == undefined) || (g == undefined) || (b == undefined))
    {
      // set default color to blue
      this.r = 0.0;
           this.g = 0.0;
      this.b = 1.0;
    }
    else
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
  }

  RGB.NO_HUE = -1;

  /*
     converts rgb to hsv, sets saturation to intensity, then hsv to rgb
     @param intensity for calculating RGB
  */
  RGB.prototype.rgbToHsvToRgb = function(intensity)
  {
    // rgb to hsv
    var max = Math.max(this.r, Math.max(this.g, this.b));
    var min = Math.min(this.r, Math.min(this.g, this.b));
    var delta = max - min;

    var h = 0;
    var s = 0;
    var v = max;

    if ((max != 0) || (max != 0.0))
    {
        s = delta / max;
    }
    else
    {
        s = 0.0;
    }

    if ((s == 0) || (s == 0.0))
    {
        h = RGB.NO_HUE;
    }
    else
```

```
{
    if (this.r == max)
    {
        h = (this.g - this.b) / delta;
    }
    else if (this.g == max)
    {
        h = 2 + (this.b - this.r) / delta;
    }
    else if (this.b == max)
    {
        h = 4 + (this.r - this.g) / delta;
    }

    h *= 60.0;

    if (h < 0)
    {
        h += 360;
    }

    h /= 360.0;
}

    // hsv to rgb
    var i;
    var aa = 0.0;
    var bb = 0.0;
    var cc = 0.0;
    var f = 0.0;

    s = intensity;

    if (s == 0)    // GrayScale
    {
        this.r = this.g = this.b = v;
    }
    else
    {
        if (h == 1.0)
        {
            h = 0;
        }

        h *= 6.0;
        i = Math.floor(h);
        f = h - i;
        aa = v * (1 - s);
        bb = v * (1 - (s * f));
        cc = v * (1 - (s * (1 - f)));
        switch(i)
        {
            case 0:
                            this.r = v;
                            this.g = cc;
                            this.b = aa;
                            break;
            case 1:
                            this.r = bb;
                            this.g = v;
                            this.b = aa;
                            break;
            case 2:
                            this.r = aa;
                            this.g = v;
                            this.b = cc;
                            break;
            case 3:
                            this.r = aa;
                            this.g = bb;
                            this.b = v;
                            break;
            case 4:
                            this.r = cc;
                            this.g = aa;
                            this.b = v;
```

```
                        break;
            case 5:
                        this.r = v;
                        this.g = aa;
                        this.b = bb;
                        break;
        }
     }
}

/*
   gets red component value ranging from 0 to 1
   return red component value ranging from 0 to 1
*/
RGB.prototype.getR = function()
{
  return this.r;
}

/*
     gets green component value ranging from 0 to 1
     return green component value ranging from 0 to 1
*/
RGB.prototype.getG = function()
  {
    return this.g;
}

/*
     gets blue component value ranging from 0 to 1
     return blue component value ranging from 0 to 1
*/
RGB.prototype.getB = function()
  {
    return this.b;
}

/*
   gets red component from 0 to 255 range
   @return red component from 0 to 255 range
*/
RGB.prototype.getRIn255Scale = function()
  {
    return Math.round(this.r * 255);
}

/*
     gets green component from 0 to 255 range
     @return green component from 0 to 255 range
*/
RGB.prototype.getGIn255Scale = function()
{
    return Math.round(this.g * 255);
}

/*
     gets blue component from 0 to 255 range
     @return blue component from 0 to 255 range
*/
RGB.prototype.getBIn255Scale = function()
  {
  return Math.round(this.b * 255);
}

/*
   to string
   @return string representation of the RGB instance
*/
RGB.prototype.toString = function()
{
  var string = "RGB(" +
               this.r + ", " +
               this.g + ", " +
               this.b +
               ")";
```

```
    return string;
  }

  /*
     creats a RGB instance with the specified r, g, b components
     @param r red component in floating point ranging from 0, to 1
        @param g green component in floating point ranging from 0, to 1
     @param b blue component in floating point ranging from 0, to 1component
  */
  function createRGB(r, g, b)
  {
    var rgb = new RGB(0, 0, 1);
    rgb.rgbToHsvToRgb(0.3);

    return rgb.getBIn255Scale();

  }
// END class RGB

// BEGIN class Group
  /*
     increments global variable "groupLevel"
     PostCondition global variable "groupLevel" is incremented
  */
  function beginGroup()
  {
    ++groupLevel;

    return "groupLevel: " + groupLevel;
  }

  /*
       decrements global variable "groupLevel"
       PostCondition global variable "groupLevel" is decremented
  */
  function endGroup()
    {
      --groupLevel;

      return "groupLevel: " + groupLevel;
  }
// END class Group

// BEGIN class Transform
  /*
     increments global variable "groupLevel", and pushed a transformation matrix
     into the transformArray.
     @param transformX translate along the x-axis
     @param transformY translate along the y-axis
     @param transformZ translate along the z-axis
     @param scaleX x scaling factor
     @param scaleY y scaling factor
     @param scaleZ z scaling factor
     @param rotateX x component of the rotation vection
     @param rotateY y component of the rotation vection
     @param rotateZ z component of the rotation vection
     @param rotateAngle angle in radians to be rotated
  */
  function createTransform(transformX, transformY, transformZ, scaleX, scaleY, scaleZ,
  rotateX, rotateY, rotateZ, rotateAngle)
  {
    ++groupLevel;


    var matrix = new TransformationMatrix();

    matrix.setMatrixTranslation(transformX, transformY, transformZ);
    matrix.setMatrixScaling(scaleX, scaleY, scaleZ);
    matrix.setMatrixRotation(rotateX, rotateY, rotateZ, rotateAngle);
    transformArray.push(matrix);

    var string = "groupLevel: " + groupLevel +
                 "transformArray.length: " + transformArray.length +
                 "transformArray[transformArray.length-1]:" +
                 transformArray[transformArray.length-1].toString();
    return string;
```

```
    return "groupLevel: " + groupLevel;
  }

  /*
      decrements global variable "groupLevel", and popps a transformation matrix
      from the transformArray.
  */
  function endTransform()
    {
      --groupLevel;
      transformArray.pop();

      return "groupLevel: " + groupLevel;
  }
// END class Transform
]]>
</msxsl:script>

<!-- output html, head, and body tag when root element is matched -->
<xsl:template match="/">
    <xsl:apply-templates />
</xsl:template>

<xsl:template match="X3D">
  <html xmlns:v="urn:schemas-microsoft-com:vml">
      <head>
          <title>Box tag from X3D to VML</title>
              <object id="VMLRender" classid="CLSID:10072CEC-8CC1-11D1-986E-00A0C955B42E">
              </object>
            <style>
              v\:* {behavior: url(#VMLRender)}
            </style>
       </head>
       <body>
         <center>
           <form>
           <table>
           <tr>
               <td></td>
               <td align="center"><input type="button" value="   move up   "
                   onclick="parent.moveUp()"/></td>
               <td></td>
               <td></td>
               <td></td>
               <td align="center"><input type="button" value="rotate up"
                  onclick="parent.rotateUp()"/></td>
               <td></td>
               </tr>

               <tr>
               <td><input type="button" value="move left"
                   onclick="parent.moveLeft()"/></td>
               <td><input type="button" value="     move down     "
                   onclick="parent.moveDown()"/></td>
               <td><input type="button" value="move right"
                   onclick="parent.moveRight()"/></td>
               <td width="20"></td>
               <td><input type="button" value="rotate left"
                   onclick="parent.rotateLeft()"/></td>
               <td><input type="button" value="rotate down"
                    onclick="parent.rotateDown()"/></td>
               <td><input type="button" value="rotate right"
                    onclick="parent.rotateRight()"/></td>
               </tr>

               <tr>
               <td colspan="7" height="20"></td>
               </tr>

               <tr>
               <td></td>
               <td align="center"><input type="button" value="move backward"
                   onclick="parent.moveBackward()"/></td>
               <td colspan="3"></td>
               <td align="center"><input type="button" value="    back    "
```

```
                    onclick="parent.back()"
                    title="go back to the previous movement"/></td>
                <td></td>
                </tr>

                <tr>
                <td></td>
                <td align="center"><input type="button" value="move forward"
                    onclick="parent.moveForward()"/></td>
                <td colspan="3"></td>
                <td align="center"><input type="button" value="forward"
                    onclick="parent.forward()"
                    title="go forward to the next movement"/></td>
                <td></td>
                </tr>
            </table>

            </form>
            <br/>
              <br/>
              <br/>
              <br/>
              <br/>
              <br/>
              <br/>
              <!-- apply histogram templates when reads a histogram tag  -->
              <xsl:apply-templates/>
            </center>
      </body>
  </html>
</xsl:template>

<!-- applies templates and parse the long polygon string when a Scene tag is matched -->
<xsl:template match="Scene">
  <xsl:apply-templates />
<xsl:variable name="endScene" select="project:endScene()" />

    <xsl:call-template name="parse-polygonString">
      <xsl:with-param name="paramString" select="$endScene"/>
    </xsl:call-template>

</xsl:template>

<!-- applies templates when a Group tag is matched -->
<xsl:template match="Group">
  <xsl:variable name="beginGroup" select="project:beginGroup()" />
  <xsl:apply-templates />
  <xsl:variable name="endGroup" select="project:endGroup()" />
</xsl:template>

<!-- gets translation, scale, and rotation attribute values when a Transform tag is
matched -->
<xsl:template match="Transform">

<xsl:variable name="scaleAttributes" select="@scale"/>
  <xsl:variable name="normalizeScaleAttributes"
       select="concat(normalize-space($scaleAttributes), ' ')"/>

  <xsl:variable name="scaleX" select="substring-before($normalizeScaleAttributes, ' ')"/>
  <xsl:variable name="scaleRest" select="substring-after($normalizeScaleAttributes, '
')"/>
  <xsl:variable name="scaleY" select="substring-before($scaleRest, ' ')"/>
  <xsl:variable name="scaleZ" select="substring-after($scaleRest, ' ')"/>


  <xsl:variable name="rotationAttributes" select="@rotation"/>
  <xsl:variable name="normalizeRotationAttributes"
       select="concat(normalize-space($rotationAttributes), ' ')"/>

  <xsl:variable name="rotationX" select="substring-before($normalizeRotationAttributes, '
')"/>
  <xsl:variable name="rotationRest" select="substring-after($normalizeRotationAttributes,
' ')"/>
  <xsl:variable name="rotationY" select="substring-before($rotationRest, ' ')"/>
  <xsl:variable name="rotationRest1" select="substring-after($rotationRest, ' ')"/>
```

```
  <xsl:variable name="rotationZ" select="substring-before($rotationRest1, ' ')"/>
  <xsl:variable name="radianAngle" select="substring-after($rotationRest1, ' ')"/>

  <xsl:variable name="translationAttributes" select="@translation"/>
  <xsl:variable name="normalizeTranslationAttributes"
       select="concat(normalize-space($translationAttributes), ' ')"/>

  <xsl:variable name="translationX"
       select="substring-before($normalizeTranslationAttributes, ' ')"/>
  <xsl:variable name="translationRest"
       select="substring-after($normalizeTranslationAttributes, ' ')"/>
  <xsl:variable name="translationY" select="substring-before($translationRest, ' ')"/>
  <xsl:variable name="translationZ" select="substring-after($translationRest, ' ')"/>

  <xsl:variable name="createTransform"
       select="project:createTransform($translationX, $translationY, $translationZ,
                                       $scaleX, $scaleY, $scaleZ,
                                       $rotationX, $rotationY, $rotationZ, $radianAngle)"
/>



  <xsl:apply-templates />
  <xsl:variable name="endTransform" select="project:endTransform()" />
</xsl:template>

<!-- a shape might have either one Box, Cone, Cylinder, or Sphere child node -->
<xsl:template match="Shape">
       <xsl:apply-templates />
    <xsl:variable name="endShape" select="project:endShape()" />

</xsl:template>

<!-- applied templates when an Appearance tag is matched -->
<xsl:template match="Appearance">
  <xsl:apply-templates />
</xsl:template>

<!-- gets color attribute values -->
<xsl:template match="Material">
   <xsl:variable name="diffuseColor" select="@diffuseColor"/>
   <xsl:variable name="r" select="substring-before($diffuseColor, ' ')"/>
   <xsl:variable name="rest" select="substring-after($diffuseColor, ' ')"/>
   <xsl:variable name="g" select="substring-before($rest, ' ')"/>
   <xsl:variable name="b" select="substring-after($rest, ' ')"/>
   <xsl:variable name="setColor" select="project:setColor($r, $g, $b)" />
</xsl:template>


<!-- gets the box's dimensions from size attribute  -->
<xsl:template match="Box">
  <!-- gets box's size from box tag -->
  <xsl:variable name="boxDim" select="@size"/>
  <xsl:variable name="x" select="substring-before($boxDim, ' ')"/>
  <xsl:variable name="rest" select="substring-after($boxDim, ' ')"/>
  <xsl:variable name="y" select="substring-before($rest, ' ')"/>
  <xsl:variable name="z" select="substring-after($rest, ' ')"/>
  <xsl:variable name="createBox" select="project:createBox($x, $y, $z)" />

</xsl:template>

<!-- gets the cylinder's height and radius -->
<xsl:template match="Cylinder">
  <xsl:variable name="height" select="@height"/>
  <xsl:variable name="normalizedHeight" select="concat(normalize-space($height), ' ')"/>
  <xsl:variable name="radius" select="@radius"/>
  <xsl:variable name="normalizedRadius" select="concat(normalize-space($radius), ' ')"/>
  <xsl:variable name="createCylinder"
       select="project:createCylinder($normalizedHeight, $normalizedRadius)" />
</xsl:template>

<!-- gets the cone's height and bottomRadius -->
<xsl:template match="Cone">
  <xsl:variable name="height" select="@height"/>
  <xsl:variable name="normalizedHeight" select="concat(normalize-space($height), ' ')"/>
  <xsl:variable name="bottomRadius" select="@bottomRadius"/>
```

```
  <xsl:variable name="normalizedBottomRadius" select="concat(normalize-
space($bottomRadius), ' ')"/>
  <xsl:variable name="createCone"
       select="project:createCone($normalizedHeight, $normalizedBottomRadius)" />
</xsl:template>

<!-- gets the sphere's radius -->
<xsl:template match="Sphere">
  <xsl:variable name="radius" select="@radius"/>
  <xsl:variable name="normalizedRadius" select="concat(normalize-space($radius), ' ')"/>
  <xsl:variable name="createSphere" select="project:createSphere($normalizedRadius)" />
</xsl:template>

<!-- parses a polygon's string  -->
<xsl:template name="parse-polygonString">
  <xsl:param name="paramString"/>
  <xsl:variable name="normalizedParam" select="concat(normalize-space($paramString), '
')"/>


  <xsl:variable name="x0" select="substring-before($normalizedParam, ' ')"/>
  <xsl:variable name="rest1" select="substring-after($normalizedParam, ' ')"/>
  <xsl:variable name="y0" select="substring-before($rest1, ' ')"/>
  <xsl:variable name="rest2" select="substring-after($rest1, ' ')"/>
  <xsl:variable name="x1" select="substring-before($rest2, ' ')"/>
  <xsl:variable name="rest3" select="substring-after($rest2, ' ')"/>
  <xsl:variable name="y1" select="substring-before($rest3, ' ')"/>
  <xsl:variable name="rest4" select="substring-after($rest3, ' ')"/>
  <xsl:variable name="x2" select="substring-before($rest4, ' ')"/>
  <xsl:variable name="rest5" select="substring-after($rest4, ' ')"/>
  <xsl:variable name="y2" select="substring-before($rest5, ' ')"/>
  <xsl:variable name="rest6" select="substring-after($rest5, ' ')"/>
  <xsl:variable name="x3" select="substring-before($rest6, ' ')"/>
  <xsl:variable name="rest7" select="substring-after($rest6, ' ')"/>
  <xsl:variable name="y3" select="substring-before($rest7, ' ')"/>
  <xsl:variable name="rest8" select="substring-after($rest7, ' ')"/>

  <xsl:variable name="r1" select="substring-before($rest8, ' ')"/>
  <xsl:variable name="rest9" select="substring-after($rest8, ' ')"/>
  <xsl:variable name="g1" select="substring-before($rest9, ' ')"/>
  <xsl:variable name="rest10" select="substring-after($rest9, ' ')"/>
  <xsl:variable name="b1" select="substring-before($rest10, ' ')"/>
  <xsl:variable name="rest11" select="substring-after($rest10, ' ')"/>
  <xsl:variable name="r2" select="substring-before($rest11, ' ')"/>
  <xsl:variable name="rest12" select="substring-after($rest11, ' ')"/>
  <xsl:variable name="g2" select="substring-before($rest12, ' ')"/>
  <xsl:variable name="rest13" select="substring-after($rest12, ' ')"/>
  <xsl:variable name="b2" select="substring-before($rest13, ' ')"/>
  <xsl:variable name="rest14" select="substring-after($rest13, ' ')"/>

  <xsl:call-template name="drawPolygon">
    <xsl:with-param name="x0" select="$x0"/>
    <xsl:with-param name="y0" select="$y0"/>
    <xsl:with-param name="x1" select="$x1"/>
    <xsl:with-param name="y1" select="$y1"/>
    <xsl:with-param name="x2" select="$x2"/>
    <xsl:with-param name="y2" select="$y2"/>
    <xsl:with-param name="x3" select="$x3"/>
    <xsl:with-param name="y3" select="$y3"/>
    <xsl:with-param name="r1" select="$r1"/>
    <xsl:with-param name="g1" select="$g1"/>
    <xsl:with-param name="b1" select="$b1"/>
    <xsl:with-param name="r2" select="$r2"/>
        <xsl:with-param name="g2" select="$g2"/>
    <xsl:with-param name="b2" select="$b2"/>
  </xsl:call-template>

  <!--- call recursively for $rest14 -->
  <xsl:variable name="length" select="string-length($rest14)"/>
  <xsl:choose>
    <xsl:when test="$length > 0">
      <xsl:call-template name="parse-polygonString">
      <xsl:with-param name="paramString" select="$rest14"/>
      </xsl:call-template>
    </xsl:when>
  </xsl:choose>
```

```
</xsl:template>

<!-- draws a polygon with 4 sides -->
<xsl:template name="drawPolygon">
  <xsl:param name="x0"/>
  <xsl:param name="y0"/>
  <xsl:param name="x1"/>
  <xsl:param name="y1"/>
  <xsl:param name="x2"/>
  <xsl:param name="y2"/>
  <xsl:param name="x3"/>
  <xsl:param name="y3"/>
  <xsl:param name="r1"/>
    <xsl:param name="g1"/>
  <xsl:param name="b1"/>
  <xsl:param name="r2"/>
    <xsl:param name="g2"/>
  <xsl:param name="b2"/>


<xsl:text disable-output-escaping="yes">&lt;v:polyline points="</xsl:text>
  <xsl:value-of select="$x0"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y0"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$x1"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y1"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$x2"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y2"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$x3"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y3"/>pt,
  <xsl:value-of select="$x0"/>pt,
  <xsl:text disable-output-escaping="no"> </xsl:text>
  <xsl:value-of select="$y0"/>pt
  <xsl:text disable-output-escaping="yes">" fillcolor="rgb(</xsl:text>
  <xsl:value-of select="$r1"/>,
  <xsl:value-of select="$g1"/>,
  <xsl:value-of select="$b1"/>
  <xsl:text disable-output-escaping="yes">)"&gt;</xsl:text>
  <xsl:text disable-output-escaping="yes">&lt;v:stroke on="false"/></xsl:text>
  <xsl:text disable-output-escaping="yes">&lt;v:fill method="linear sigma" angle="180"
       type="gradient" color2="rgb(</xsl:text>
  <xsl:value-of select="$r2"/>,
    <xsl:value-of select="$g2"/>,
    <xsl:value-of select="$b2"/>
  <xsl:text disable-output-escaping="yes">)"/&gt;</xsl:text>
  <xsl:text disable-output-escaping="yes">&lt;/v:polyline&gt;</xsl:text>

</xsl:template>

</xsl:stylesheet>
```

APPENDIX E

Source code for the “X3dToVml.html” file.

```
<html>

<script type="text/javascript">
// Load the X3D input file
var xml = new ActiveXObject("Microsoft.XMLDOM");
xml.async = false;
xml.load("threeBoxes.xml");
```

```
// Load the XSLT file
var xsl = new ActiveXObject("Microsoft.XMLDOM");
xsl.async = false;
xsl.load("x3dTovml.xsl");

// temporily stores nodes removed from "back" events
var forwardNode = xml.createElement("forward");

// max Transform tags could be added, combine them if exceed this number
var historyMax = 5;

// number of movement history so far, cannot be greater than historyMax
var historyCount = 0;

// array stores group nodes
var groupMap = {};

// flag if has called addDoNothingRotateXandYTags() function
var hasAddedDoNothingRotateXandYTags = "false";

// stores rotation matrix of combined Transform tag
var currentMatrix = new TransformationMatrix();


// BEGIN class Vector
  /*
     default constructor
  */
  function Vector(x, y, z)
  {
    this.vectorArray = new Array(x, y, z);
  }

  /*
     normalizes vector
  */
  Vector.prototype.normalize = function()
  {
    var magnitude = this.getMagnitude();

    this.vectorArray[0] /= magnitude;
    this.vectorArray[1] /= magnitude;
    this.vectorArray[2] /= magnitude;
  }



  /*
     gets x value
     @return x value
  */
  Vector.prototype.getX = function()
  {
    return this.vectorArray[0];
  }

  /*
     gets y value
     @return y value
  */
  Vector.prototype.getY = function()
    {
        return this.vectorArray[1];
  }

  /*
     gets z
     @return z value
  */
  Vector.prototype.getZ = function()
    {
        return this.vectorArray[2];
  }

  /*
```

```
    gets magnitude
    @return magnitude
  */
  Vector.prototype.getMagnitude = function()
  {
      var magnitude = this.vectorArray[0]*this.vectorArray[0] +
                      this.vectorArray[1]*this.vectorArray[1] +
                      this.vectorArray[2]*this.vectorArray[2];

    magnitude = Math.sqrt(magnitude);

    return magnitude;
  }


  /*
     to string
     @return string representation of this vector
  */
  Vector.prototype.toString = function()
  {
    var string = "Vector("  +
                 this.vectorArray[0] + ", " +
                 this.vectorArray[1] + ", " +
                 this.vectorArray[2] +
                 ")";

    return string;
  }

// END class Vector



// BEGIN class TransformationMatrix
  /*
     default constructor
  */
    function TransformationMatrix()
  {
      var row0 = new Array(1, 0, 0, 0);
        var row1 = new Array(0, 1, 0, 0);
        var row2 = new Array(0, 0, 1, 0);
        var row3 = new Array(0, 0, 0, 1);
        this.matrix = new Array(row0, row1, row2, row3);
    }

  /*
       multiplies the parameter matrix with this matrix (this = paramMatrix * this)
       @param paramMatrix matrix to be multiplied with
    */
    TransformationMatrix.prototype.matrixPostMultiply = function(paramMatrix)
    {
      var temp = new TransformationMatrix();

      for (var row = 0; row < 4; ++row)
      {
        for (var column = 0; column < 4; ++column)
        {
          temp.matrix[row][column] = paramMatrix.matrix[row][0]*this.matrix[0][column] +
                                     paramMatrix.matrix[row][1]*this.matrix[1][column] +
                                     paramMatrix.matrix[row][2]*this.matrix[2][column] +
                                     paramMatrix.matrix[row][3]*this.matrix[3][column];
        }
      }

      for (var row = 0; row < 4; ++row)
      {
        for (var column = 0; column < 4; ++column)
        {
          this.matrix[row][column] = temp.matrix[row][column];
        }
      }
    }
```

```
  /*
      calculates the matrix is rotated through an angle radianAngle around axis (x y z)
vector
       @param x x compounent of the rotation axis
       @param y y compounent of the rotation axis
       @param z z compounent of the rotation axis
    */
    TransformationMatrix.prototype.setMatrixRotation = function(x, y, z, radianAngle)
    {

      var u = new Vector(x, y, z);

      u.normalize();

      var uX = u.getX();
      var uY = u.getY();
      var uZ = u.getZ();

        var c = Math.cos(radianAngle);
      var s = Math.sin(radianAngle);
      var temp = new TransformationMatrix();

      temp.matrix[0][0] = (1 - c)*uX*uX + c;
        temp.matrix[0][1] = (1 - c)*uX*uY - s*uZ;
        temp.matrix[0][2] = (1 - c)*uX*uZ + s*uY;

        temp.matrix[1][0] = (1 - c)*uX*uY + s*uZ;
        temp.matrix[1][1] = (1 - c)*uY*uY + c;
        temp.matrix[1][2] = (1 - c)*uY*uZ - s*uX;

        temp.matrix[2][0] = (1 - c)*uX*uZ - s*uY;
        temp.matrix[2][1] = (1 - c)*uY*uZ + s*uX;
        temp.matrix[2][2] = (1 - c)*uZ*uZ + c;

        this.matrixPostMultiply(temp);
    }

  /*
      returns a vector and angle representation of the rotation matrix
      @return a string containing a vector and angle in radians
  */
    TransformationMatrix.prototype.getUnitVectorAndRadianFormatInString = function()
    {
      var alpha = this.matrix[0][0] + this.matrix[1][1] + this.matrix[2][2] - 1;

      var b1 = this.matrix[2][1] - this.matrix[1][2];
      var b2 = this.matrix[0][2] - this.matrix[2][0];
      var b3 = this.matrix[1][0] - this.matrix[0][1];
      var beta = Math.sqrt(b1*b1 + b2*b2 + b3*b3);

    var uX = (this.matrix[2][1] - this.matrix[1][2]) / beta;
    var uY = (this.matrix[0][2] - this.matrix[2][0]) / beta;
    var uZ = (this.matrix[1][0] - this.matrix[0][1]) / beta;
    var u = new Vector(uX, uY, uZ);

    u.normalize();
    uX = u.getX();
    uY = u.getY();
    uZ = u.getZ();

    var theta = Math.atan2(beta, alpha);
    var string = uX + " " + uY + " " + uZ + " " + theta;

    return string;
    }

  /*
     to string
     @return string representation of the matrix
  */
    TransformationMatrix.prototype.toString = function()
    {
      var string = "Matrix {";

      for (var i = 0; i < 4; ++i)
        {
```

```
          string += "[ ";

          for (var j = 0; j < 4; ++j)
            {
              string += this.matrix[i][j] + " ";
            }

            string += "]";
        }

        string += " }";

        return string;
    }
// END class TransformationMatrix



/*
   applies the XSLT file to the input xml file
*/
function transformNodes()
{
   vmlFrame.document.write(xml.transformNode(xsl));
}

/*
   calls handleUSEDEF() function
*/
handleUSEDEF();

/*
   replaces all USE attribute with their corresponding DEF nodes
*/
function handleUSEDEF()
{
  var groupNodeList = xml.getElementsByTagName("Group");

  var defName = "";


  for (var i = 0; i < groupNodeList.length; ++i)
  {
     defName = groupNodeList.item(i).getAttribute("DEF");

     if (defName != null)
     {
        groupNodeList.item(i).removeAttribute("DEF");
        groupMap[defName] = groupNodeList.item(i).cloneNode(true);
     }
  }

  replaceUSE(xml.documentElement.firstChild);
}

/*
   replace all Group node with USE attribute with their corresponding DEF nodes
   @param node current node will be replaced with its corresponding DEF nodes
*/
function replaceUSE(node)
{

   var useName = node.getAttribute("USE");

   var tempNode = "";

   if (useName != null)
   {
     node.removeAttribute("USE");

     tempNode = groupMap[useName].cloneNode(true);

     node.parentNode.replaceChild(tempNode, node);
```

```
   }

   if (node.hasChildNodes())
   {
      replaceUSE(node.firstChild);
   }


   if (node.nextSibling != null)
   {
      replaceUSE(node.nextSibling);
   }

}


/*
    combines the 5th Transform node with the 6th Transform node
*/
function combineTransformNodes()
{
  var sceneNode = xml.documentElement.firstChild;
  var _5thNode = sceneNode.firstChild.firstChild.firstChild.firstChild.firstChild;

  var _5thNodeRotationValue = _5thNode.getAttribute("rotation");

  var _5thNodeRotationValueList = _5thNodeRotationValue.split(" ");


  var _6thNode = _5thNode.firstChild;

  if ((_5thNodeRotationValueList[0] == "1") || (_5thNodeRotationValueList[1] == "1"))
  {
     var _5thNodeRotationX = _5thNodeRotationValueList[0] - 0;
     var _5thNodeRotationY = _5thNodeRotationValueList[1] - 0;
     var _5thNodeRotationZ = _5thNodeRotationValueList[2] - 0;
     var _5thNodeRotationRadian = _5thNodeRotationValueList[3] - 0;

     currentMatrix.setMatrixRotation(_5thNodeRotationX, _5thNodeRotationY,
                                     _5thNodeRotationZ, _5thNodeRotationRadian);

     var newRotationValue1 = currentMatrix.getUnitVectorAndRadianFormatInString();

     _6thNode.setAttribute("rotation", newRotationValue1);

  }
  else
  {
     var _5thNodeTranslationValue = _5thNode.getAttribute("translation");

     _5thNodeTranslationValueList = _5thNodeTranslationValue.split(" ");

     var _6thNodeTranslationValue = _6thNode.getAttribute("translation");

     _6thNodeTranslationValueList = _6thNodeTranslationValue.split(" ");

     var x = _5thNodeTranslationValueList[0] - 0;   // converts to a number
     var y = _5thNodeTranslationValueList[1] - 0;
     var z = _5thNodeTranslationValueList[2] - 0;

     var x1 = _6thNodeTranslationValueList[0] - 0;   // converts to a number
       var y1 = _6thNodeTranslationValueList[1] - 0;
     var z1 = _6thNodeTranslationValueList[2] - 0;

     x += x1;
     y += y1;
     z += z1;

     var newTranslationValue = x + " " + y + " " + z;

     _6thNode.setAttribute("translation", newTranslationValue);
  }

  var _4thNode = sceneNode.firstChild.firstChild.firstChild.firstChild;
  _4thNode.replaceChild(_5thNode.firstChild, _5thNode);
```

```
}


/* wraps the following tags to the child tags of the scene tag
      <Transform rotation="0 1 0 0" scale="1 1 1" translation="0 0 0">
      </Transform>
   This tag will be used for combining rotation Transform tags when calling
   combineTransformNode() function.
*/
function addDoNothingRotateXandYTags()
{
   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","0 1 0 0");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "0 0 0");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

}

/*
   moves objects 20 units to the left
*/
function moveLeft()
{

   if (hasAddedDoNothingRotateXandYTags == "false")
   {
     addDoNothingRotateXandYTags();
     hasAddedDoNothingRotateXandYTags = "true";
   }


   forwardNode = xml.createElement("forward");
   ++historyCount;


   if (historyCount > historyMax)
   {
     combineTransformNodes();
     historyCount = historyMax;
   }


   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","0 0 1 0");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "-20 0 0");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
```

```
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

   transformNodes();

   vmlFrame.document.close();
}


/*
   moves the objects 20 units up
*/
function moveUp()
{

  if (hasAddedDoNothingRotateXandYTags == "false")
   {
     addDoNothingRotateXandYTags();
     hasAddedDoNothingRotateXandYTags = "true";
   }


   forwardNode = xml.createElement("forward");
   ++historyCount;

   if (historyCount > historyMax)
   {
     combineTransformNodes();
     historyCount = historyMax;
   }

   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","0 0 1 0");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "0 -20 0");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

   transformNodes();
   vmlFrame.document.close();
}


/*
   moves the objects 20 units down
*/
function moveDown()
{

   if (hasAddedDoNothingRotateXandYTags == "false")
   {
     addDoNothingRotateXandYTags();
     hasAddedDoNothingRotateXandYTags = "true";
   }


   forwardNode = xml.createElement("forward");
   ++historyCount;

   if (historyCount > historyMax)
   {
     combineTransformNodes();
     historyCount = historyMax;
```

```
   }

   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","0 0 1 0");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "0 20 0");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

   transformNodes();
   vmlFrame.document.close();
}


/*
   moves the objects 20 units to the right
*/
function moveRight()
{

   if (hasAddedDoNothingRotateXandYTags == "false")
   {
     addDoNothingRotateXandYTags();
     hasAddedDoNothingRotateXandYTags = "true";
   }


   forwardNode = xml.createElement("forward");
   ++historyCount;

   if (historyCount > historyMax)
   {
     combineTransformNodes();
     historyCount = historyMax;
   }

   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","0 0 1 0");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "20 0 0");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

   transformNodes();
   vmlFrame.document.close();
}

/*
```

```
   moves the objects 20 units backward
*/
function moveBackward()
{

    if (hasAddedDoNothingRotateXandYTags == "false")
    {
      addDoNothingRotateXandYTags();
      hasAddedDoNothingRotateXandYTags = "true";
    }


    forwardNode = xml.createElement("forward");
    ++historyCount;

    if (historyCount > historyMax)
    {
      combineTransformNodes();
      historyCount = historyMax;
    }

   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","0 0 1 0");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "0 0 20");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);
   transformNodes();
   vmlFrame.document.close();
}

/*
   moves the objects 20 units forward
*/
function moveForward()
{

   if (hasAddedDoNothingRotateXandYTags == "false")
   {
     addDoNothingRotateXandYTags();
     hasAddedDoNothingRotateXandYTags = "true";
   }


   forwardNode = xml.createElement("forward");
   ++historyCount;

   if (historyCount > historyMax)
   {
     combineTransformNodes();
     historyCount = historyMax;
   }

   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","0 0 1 0");
```

```
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "0 0 -20");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

   transformNodes();
   vmlFrame.document.close();
}

/*
   rotates the objects to the left 0.314 radians
*/
function rotateLeft()
{

   if (hasAddedDoNothingRotateXandYTags == "false")
   {
     addDoNothingRotateXandYTags();
     hasAddedDoNothingRotateXandYTags = "true";
   }


   forwardNode = xml.createElement("forward");
   ++historyCount;

    if (historyCount > historyMax )
   {
     combineTransformNodes();
     historyCount = historyMax;
   }

   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","0 1 0 0.314");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "0 0 0");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

   transformNodes();
   vmlFrame.document.close();
}

/*
   rotates the objects to the right 0.314 radians
*/
function rotateRight()
{
    if (hasAddedDoNothingRotateXandYTags == "false")
    {
      addDoNothingRotateXandYTags();
      hasAddedDoNothingRotateXandYTags = "true";
    }

    forwardNode = xml.createElement("forward");
    ++historyCount;
```

```
    if (historyCount > historyMax)
    {
      combineTransformNodes();
      historyCount = historyMax;
    }

   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","0 1 0 -0.314");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "0 0 0");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

   transformNodes();
   vmlFrame.document.close();
}

/*
   rotates the objects up 0.314 radians
*/
function rotateUp()
{

    if (hasAddedDoNothingRotateXandYTags == "false")
    {
      addDoNothingRotateXandYTags();
      hasAddedDoNothingRotateXandYTags = "true";
    }


    forwardNode = xml.createElement("forward");
    ++historyCount;

    if (historyCount > historyMax)
    {
      combineTransformNodes();
      historyCount = historyMax;
    }

   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","1 0 0 -0.314");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "0 0 0");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

   transformNodes();
```

```
   vmlFrame.document.close();
}


/*
   rotates the objects down 0.314 radians
*/
function rotateDown()
{

   if (hasAddedDoNothingRotateXandYTags == "false")
   {
     addDoNothingRotateXandYTags();
     hasAddedDoNothingRotateXandYTags = "true";
   }


   forwardNode = xml.createElement("forward");
   ++historyCount;

     if (historyCount > historyMax)
    {
     combineTransformNodes();
     historyCount = historyMax;
    }

   vmlFrame.document.open();

   var sceneNode = xml.documentElement.firstChild;
   var sceneNodeClone = sceneNode.cloneNode(true);
   var sceneNodesList = xml.createDocumentFragment();
   sceneNodesList = sceneNodeClone.childNodes;

   var newnode=xml.createElement("Transform");
   newnode.setAttribute("rotation","1 0 0 0.314");
   newnode.setAttribute("scale", "1 1 1");
   newnode.setAttribute("translation", "0 0 0");

   for (var i = sceneNodesList.length - 1; i >= 0; --i)
   {
     newnode.appendChild(sceneNodesList.item(i));
   }

   var newSceneNode = xml.createElement("Scene");
   newSceneNode.appendChild(newnode);
   xml.documentElement.replaceChild(newSceneNode, sceneNode);

   transformNodes();
   vmlFrame.document.close();
}

/*
   goes back to the previous movement
*/
function back()
{
   if (historyCount > 0)
   {

      --historyCount;
      vmlFrame.document.open();

      var firstTransformNode = xml.documentElement.firstChild.firstChild;
      var toBeRemovedNode = firstTransformNode.cloneNode(true);

      var tempNodeList = xml.createDocumentFragment();
      tempNodeList = toBeRemovedNode.childNodes;

      var newSceneNode = xml.createElement("Scene");

      for (var i = tempNodeList.length - 1; i >= 0; --i)
      {
        newSceneNode.appendChild(tempNodeList.item(i));
      }
```

```
      forwardNode.appendChild(toBeRemovedNode.cloneNode(false));

      var sceneNode = xml.documentElement.firstChild;
      xml.documentElement.replaceChild(newSceneNode, sceneNode);

      transformNodes();
      vmlFrame.document.close();
   }
}

/*
    goes forward to the next movement
*/
function forward()
{
  if (forwardNode.hasChildNodes())
  {
    ++historyCount;

    vmlFrame.document.open();

    var oldNode = forwardNode.removeChild(forwardNode.lastChild);

    var sceneNode = xml.documentElement.firstChild;
    var sceneNodeClone = sceneNode.cloneNode(true);
    var sceneNodesList = xml.createDocumentFragment();
    sceneNodesList = sceneNodeClone.childNodes;

    for (var i = sceneNodesList.length - 1; i >= 0; --i)
    {
      oldNode.appendChild(sceneNodesList.item(i));
    }

    var newSceneNode = xml.createElement("Scene");
    newSceneNode.appendChild(oldNode);
    xml.documentElement.replaceChild(newSceneNode, sceneNode);

    transformNodes();
    vmlFrame.document.close();

  }
}

</script>

<frameset>
<frame name="vmlFrame" src="initPage.html" />
</frameset>
</html>
```