

## Table of Contents

<b>1. Introduction.....</b>	<b>3</b>
<b>2. Design.....</b>	<b>10</b>
2.1 Data Flow Diagram.....	10
2.2 Program's Design.....	11
<b>3. Requirements.....</b>	<b>20</b>
3.1 Browser.....	20
3.2 DTD file.....	20
3.3 X3D Tags Supported in This Project.....	21
3.4 Features and User Interface.....	21
3.5 The Maximum Load.....	23
3.6 Limitations.....	24
<b>4. Implementation.....</b>	<b>28</b>
4.1 The Output Document's Type.....	28
4.2 Access/Load A Source X3D File.....	29
4.3 A DOM Structure of An X3D File.....	30
4.4 Manipulate A DOM Structure of A Source X3D File.....	34
4.5 Using DOM API to Dynamically Create and Insert SVG Tags to The Output Document.....	40
4.6 Viewing Pipeline Technique.....	44
4.7 3D Object Models.....	45
4.8 Viewing Transformation.....	54
4.9 Perspective Projection.....	57
4.10 3D Geometric Transformations.....	59
4.11 Illumination Model.....	62
4.12 Gouraud Shading Model.....	65

4.13	HSV Color Model.....	68
4.14	Visible-Surface Detection.....	72
<b>5.</b>	<b>Optimization.....</b>	<b>81</b>
5.1	Optimization of The Rendering-Graphics Algorithms.....	81
5.2	Performance Hints to Speed Up JavaScript Code.....	88
<b>6.</b>	<b>Deployment.....</b>	<b>95</b>
6.1	Performance Observations on The Second Version of The Translator of Our Program.....	95
6.2	Performance Observations Among Three Versions of The Translator.....	96
6.3	Performance Observations on The Surface-Rendering Process of The Translator Version 1 and 2.....	97
6.4	Processing Time of A Surface-Rendering Process.....	98
<b>7.</b>	<b>Conclusion.....</b>	<b>99</b>
7.1	Project Achievements.....	99
7.2	Future Work.....	100
	<b>References.....</b>	<b>104</b>
	<b>Appendix A: Mathematics for Computer Graphics.....</b>	<b>106</b>
	<b>Appendix B: Source Code.....</b>	<b>108</b>

# 1. Introduction

At the present time, one cannot directly view 3D graphics on most common web browsers (Internet Explorer or Netscape) without using a plug-in. For this project we want to develop a translator so that users can view 3D graphics on common browsers without installing plug-ins. X3D (Extensible 3D) is a W3C open standard for 3D on the web and SVG (Scalable Vector Graphics) is an XML based mark-up language for describing 2D line-art graphics. SVG is currently supported by a plugin available from Adobe and supported natively by the Mozilla-SVG browser. It is likely that SVG will be supported natively in future versions of both Netscape and Internet Explorer. The goal of this project is to develop a translator from X3D to SVG that would allow web sites to exploit 3D objects without having to worry about whether the client is patient enough and competent enough to download and install a plugin. The translator outputs a document that displays a view of smooth shaded 3D objects and has features so that clients can view objects in different perspectives as it can be done in an X3D browser.

An X3D document is a tag-based document used to describe 3D objects. The following sample is an X3D document for creating a box with a size of 20 x 20 x 20.

```
<X3D>
  <Scene>
    <Shape>
      <Box size="20 20 20"/>
    </Shape>
  </Scene>
</X3D>
```

An SVG document is an XML based mark-up language for describing 2D vector graphics. The example below is an SVG application which creates a rectangle. The points attribute in this tag is used to specify the coordinates of polygon vertices and the order of the coordinates is also relevant.

```
<svg>
  <polygon points="0 0, 0 10, 10 10, 10 0"/>
</svg>
```

Our translator takes X3D as an input document and translates it into an SVG document, which is originally designed for 2D line-art graphics, to present a view of 3D scenes. The process used in our translator can be divided into two types, a translation and rendering process. The translation process is to get a parse the structure of a source X3D document, and then translate it into a corresponding structure of an SVG document to present a view of this 3D object. The rendering process involves complicated computations for drawing and shading objects.

In the translation process, we use the DOM API (Document Object Model Application Programming Interface) to traverse a tree-structure X3D source document so that we can get a parse structure of the input document. We use this to determine which primitive will be created and where it should be rendered in the scene. Typically, 3D objects are constructed as a mesh of 2D polygons. For example, a box may be created as a mesh of six polygons; a sphere may be created as a mesh of sixty-four polygons, etc. In this project, we apply the same idea for rendering 3D objects. X3D objects will be represented by `<polygon>` and other related tags supported by SVG. `<polygon>` tags and other necessary elements will be dynamically created, inserted and modified in

order to produce a correct view of 3D scenes. These tasks are done through DOM interfaces as well. More details about how we use DOM API to access and manipulate a source file and SVG nodes in the output document on the fly are described in section 4.4. Once we know which primitives are to be used and where we want to generate, the next step is to perform a rendering task.

The `<polygon>` tag can be considered as a core element we use to construct a primitive shape such as a box, a cone, a cylinder, and a sphere. To create a polygon shape in SVG, we need to provide coordinates of each polygon vertex to those tags. The coordinates we have to supply depend on which perspective we are using to view the objects. Hence, whenever users change perspectives, the coordinates must be recomputed. There is a formal mechanism in computer graphics called “the viewing pipeline” to model a 3D scene and display it on an output device in different perspectives. We apply this technique in our project to model a coordinate transformation for generating and rendering a correct view of 3D objects in different orientations. More details about the viewing pipeline are described in section 4.6.

Supplying coordinates of each polygon vertex to SVG can only display a wire-frame representation of 3D objects. However, we also want to present objects as smooth surfaces and at the same time try to reduce or eliminate polygon edge boundaries as much as possible. This can be done by interpolating a shading pattern across a polygon surface. There are many lighting or illumination models in computer graphics that we can exploit

to render graphics. In this project, we decided to use the “Gouraud Shading” model to perform linear interpolation on a polygon surface. Fortunately, SVG supports a `<linearGradient>` tag which we will use to apply linear shading to the interior of each polygon. This tag performs a smooth color transition from one shade to another. It also requires us to supply a starting and stopping color. Thus, we have to calculate color intensity along a polygon edge from one side to another and provide it to the `<linearGradient>` tag as a starting and stopping color. The intensity along the polygon edge has different values in different perspectives. As a result, whenever users change perspectives, the intensity needs to be recomputed. We leave other details about how we calculate the intensity and apply the “Gouraud Shading” model in this project to section 4.12.

As already mentioned, 3D objects are constructed as a mesh of 2D polygons to display a correct view of 3D objects on a computer screen. The polygons must be rendered in the correct order. For example, the back face of a box should be drawn on the screen before the front face of a box. Otherwise, we will get a box with mixed up faces. There are numerous computer graphics algorithms for solving this problem. However, we cannot simply apply these algorithms to efficiently obtain the correct order for drawing polygons because most of the known algorithms are designed for rendering objects at a pixel level, not a polygon level as we do in our translator. We cannot internally describe shapes, polygons, or colors of objects with pixel arrays by using SVG as can be done in common graphics-programming packages like OpenGL. So in our translator, we arrange an order

to draw polygons by using our own invented technique similar to a depth sorting method. The technique we use is that each polygon must be rendered according to its depth. The deeper ones should be drawn before the closer one as in the painter's algorithm. We will discuss how we arrange the order for rendering polygons in section 4.14.

Our translator is developed using ECMAScript [ECMA99] or JavaScript. Most applications implemented by JavaScript are quite trivial and simple. Since JavaScript is an interpreted language, a program running in JavaScript interpreter is usually slower than a program running in compiled languages such as C, C++, or Java. Normally, JavaScript has been used for small-scale programs such as web applications to interact with the user. A sophisticated and non-trivial application like a graphics-rendering program is not a typical task implemented in Javascript. Because the translator we developed in this project is quite a complicated application involving a lot of computational tasks especially in the rendering process, performance has become an issue that we need to be concerned about. We can boost system performance by improving JavaScript's performance and optimizing a rendering-graphics algorithm. Unfortunately, we cannot do much to control performance in JavaScript because resources in JavaScript are dynamically allocated and deallocated by the interpreter. Also JavaScript uses a garbage collector like the one in Java to free up memory space when it is no longer needed. However, there are some techniques that we can use to speed up our JavaScript code that will be described later in the section 5. So improvements to our system performance come from the optimization of the algorithm we use to generate 3D objects.

Instead of applying the original algorithm directly, i.e., drawing every single polygon on the screen whether or not it appears visible or invisible to a viewer, we selectively draw only the front face, which is visible to a viewer, not the back face, which is invisible to the viewer. We also try optimizing our program by using a hidden-surface removal algorithm to avoid drawing any unseen front-face polygons which are hidden by other surfaces. We have developed three versions of the translator to compare their performance. The first version, which is not an optimized version, will draw every polygon on the screen whether or not it is a back face or a front face. The second version does not draw any back face on the screen because users will not be able to see this polygon anyway. The last version applies a hidden-surface removal algorithm to render only a seen front-face polygon, not an unseen front-face polygon. After experimenting with all three versions, we found that the second version can improve the system performance by 30-40 %. The last version does not always speed up the translator. In fact, it drops the system performance more than 100 % as a result of the overhead used to check the hidden surfaces. In the section 5, we will explain more about the hidden-surface removal algorithm. Also in the section 6, we present all test cases that we have experimented with to show that this algorithm significantly reduces the performance.

In this paper, we show pseudo code and JavaScript code along with the explanation. To avoid confusion between a comment and pseudo code, any text between a `//` and the end of a line is treated as a comment. Any text between the characters `/*` and `*/` is also treated as a comment. Any text between `/@` and `@/` is considered to be pseudo code.



```
function demo( )
{
    // Text in this line is a comment for test( )
    test( )

    /*
    * Text in this line is a comment for test1( )
    */
    test1( )

    /@ This is pseudo code@/
}
```

## 2. Design

### 2.1 Data Flow Diagram

Below is a data flow diagram showing processes in this application along with inputs and outputs.

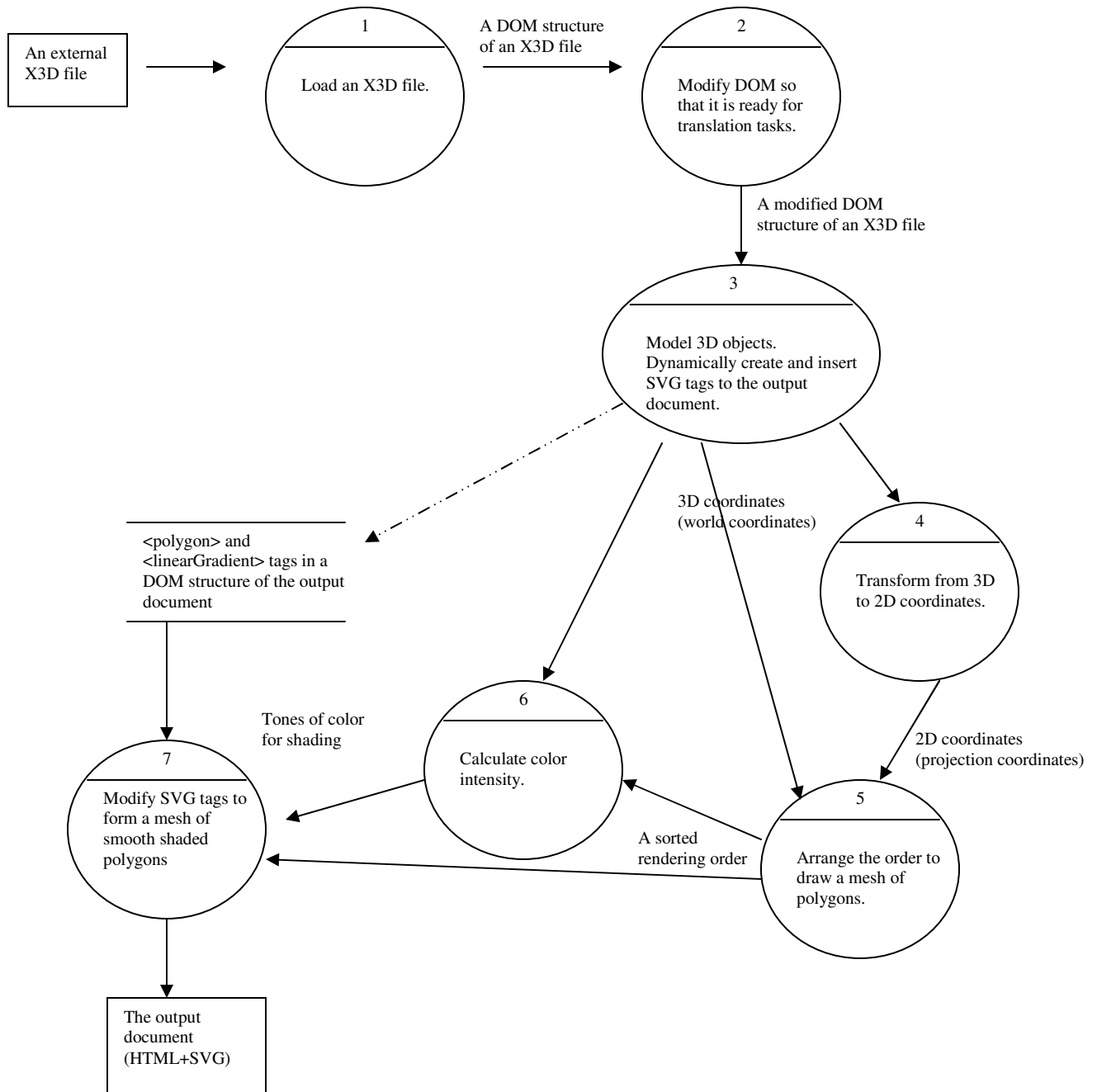
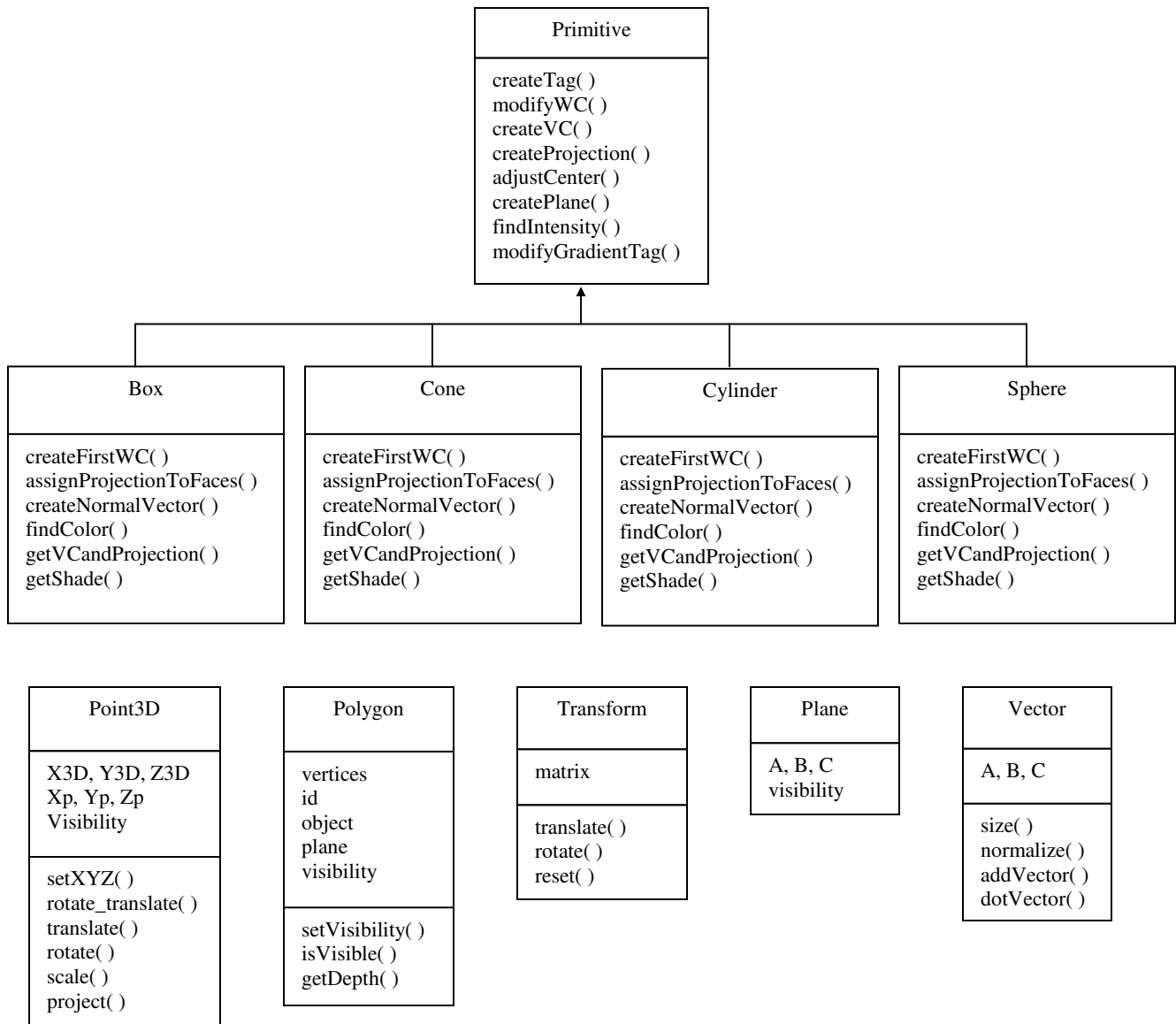


Figure 2-1 Data flow diagram of this application.

## **2.2 Program Design**

This application is developed using an object-oriented approach. We created ten classes for this application, one superclass- Primitive-, four subclasses -Box, Cone, Cylinder, and Sphere-, five helper classes -Point3D, Polygon, Transform, Plane, and Vector. The Primitive superclass has general methods for its inheritance classes to perform general translation and rendering operations. Each subclass has a constructor and methods to model its object and perform different tasks. A Box class is to model and render a box object; a Cone class is to model and render a cone object; a Cylinder class is to model and render a cylinder object; and a Sphere class is to model and render a sphere object. The five helper classes do not have any relation to the other classes. They are used along with the other classes and functions in the program. Below is a UML diagram showing all classes with their inheritance relations, if any, as well as their methods.



**Figure 2-2** UML diagram of all classes.

In our application, we factorized recurring code segments, which could create a risk to maintenance. As an example of where we use factorization after an object has been modeled, the process to compute projected positions is activated by calling a single method `getVCandProjection()`, instead of invoking a sequence of five methods. Similarly, the `getShade()` method for calculating the tone of an object's color for

shading polygons factorizes a recurring sequence of four methods. Every primitive subclass -Box, Cone, Cylinder, and Sphere- has these factorizing methods to encapsulate code sequences as follows.

```

Subclass.getVCandProjection()
{
    this.createVC();
    this.createProjection();
    this.assignProjectionToFaces();
    this.adjustCenter();
    this.createPlane();
}

Subclass.getShade()
{
    this.createNormalVector();
    this.findIntensity();
    this.findColor();
    this.modifyGradientTag();
}

```

So only one method is needed to perform a viewing transformation to get x-y projections and another one method is used to calculate color intensity for linearly interpolating across the polygon surfaces.

The following is a list of all classes including their properties and methods.

## Primitive Class

Constructor Summary	
Primitive(thisNode, appearanceNode, id)	
Create a primitive object with a specified X3D primitive and <appearance> node, and an ID. These parameters are given by its subclasses.	
Property Summary	
node	An X3D primitive node for modeling an object.
appearanceNode	An X3D appearance node for specifying a color.
id	An object's ID.
parent	A parent node.
center1	A center point of the object before transformation.
center2	A center point of the object after transformation.

num_vertices	A number of vertices of an object.
num_face	A number of polygons forming an object.
rgb	A color of the object in RGB format.
wc	An array of Point3D objects for describing world coordinates.
vc	An array of Point3D objects for describing viewing coordinates.
projection	An array of Point3D objects for describing projection coordinates.
plane	An array of Plane objects for describing polygon surfaces.
face	An array containing a set of vertices for each surface.
avgNormalVectors	An array of Vector objects for describing an average normal vector of each vertex.
intensity	An array containing light intensity of each vertex.
color_left	An array containing an RBG color of the left ledge of a polygon.
color_right	An array containing an RBG color of the right ledge of a polygon.

<b>Method Summary</b>	
createTag()	Create <polygon> and <linearGradient> tags and insert to the root SVG node.
modifyWC()	Calculate new transformed world coordinates.
createVC()	Convert world coordinates to viewing coordinates.
createProjection()	Calculate projected positions.
adjustCenter()	Calculate a new position of a center of the object.
createPlane()	Create a plane vector representing each surface.
findIntensity()	Calculate light intensity of each vertex.
modifyGradientTag()	Assign a starting and stopping color to <linearGradient> tags.

## Box Class

Constructor Summary
<code>Box(thisNode, appearanceNode, id)</code> Create a box object with a specified X3D primitive and <appearance> node, and an assigned ID.
Property Summary
<code>x</code> A box's dimension in the x-axis
<code>y</code> A box's dimension in the y-axis
<code>z</code> A box's dimension in the z-axis
<code>avgI</code> An average intensity along a box's edge
Method Summary
<code>createFirstWC()</code> Model a box object.
<code>assignProjectionToFaces()</code> Assign a series of projected coordinates to each polygon.
<code>createNormalVector()</code> Calculate an average normal vector for each vertex.
<code>findColor()</code> Calculate a tone of color along the polygon edge.
<code>getVCandProjection()</code> Perform viewing and projection transformations.
<code>getShade()</code> Perform surface-rendering tasks.

## Cone Class

Constructor Summary
<code>Cone(thisNode, appearanceNode, id)</code> Create a cone object with a specified X3D primitive node, an <appearance> node, and an assigned ID.
Property Summary
<code>bottomRadius</code> A bottom radius of a cone.
<code>height</code> A height of a cone.
<code>avgI</code> An average intensity along a box's edge.
Method Summary
<code>createFirstWC()</code> Model a cone object.
<code>assignProjectionToFaces()</code> Assign a series of projected coordinates to each polygon.

<code>createNormalVector()</code> Calculate an average normal vector for each vertex.
<code>findColor()</code> Calculate a tone of color along the polygon edge.
<code>getVCandProjection()</code> Perform viewing and projection transformations.
<code>getShade()</code> Perform surface-rendering tasks.

## Cylinder Class

<b>Constructor Summary</b>
<code>Cylinder(thisNode, appearanceNode, id)</code> Create a cylinder object with a specified X3D primitive and <appearance> node, and an assigned ID.

<b>Property Summary</b>
<code>radius</code> A radius of a cylinder.
<code>height</code> A height of a cylinder.
<code>avgI</code> An average intensity along a box's edge.

<b>Method Summary</b>
<code>createFirstWC()</code> Model a cylinder object.
<code>assignProjectionToFaces()</code> Assign a series of projected coordinates to each polygon.
<code>createNormalVector()</code> Calculate an average normal vector for each vertex.
<code>findColor()</code> Calculate a tone of color along the polygon edge.
<code>getVCandProjection()</code> Perform viewing and projection transformations.
<code>getShade()</code> Perform surface-rendering tasks.

## Sphere Class

<b>Constructor Summary</b>
<code>Sphere(thisNode, appearanceNode, id)</code> Create a sphere object with a specified X3D primitive and <appearance> node, and an assigned ID.

<b>Property Summary</b>
<code>radius</code> A radius of a sphere.
<code>avgI</code> An average intensity along a box's edge.



<b>Method Summary</b>
<code>createFirstWC()</code> Model a sphere object.
<code>assignProjectionToFaces()</code> Assign a series of projected coordinates to each polygon.
<code>createNormalVector()</code> Calculate an average normal vector for each vertex.
<code>findColor()</code> Calculate a tone of color along the polygon edge.
<code>getVCandProjection()</code> Perform viewing and projection transformations.
<code>getShade()</code> Perform surface-rendering tasks.

## Point3D Class

<b>Constructor Summary</b>
<code>Point3D(x, y, z)</code> Create a point object with a specified x-y-z coordinate.

<b>Property Summary</b>
<code>X3D</code> An x coordinate in 3D.
<code>Y3D</code> A y coordinate in 3D.
<code>Z3D</code> A z coordinate in 3D.
<code>Xp</code> An x coordinate in 2D.
<code>Yp</code> A y coordinate in 2D.
<code>Zp</code> A z coordinate in 2D.
<code>visibility</code> A visibility property.

<b>Method Summary</b>
<code>setXYZ(x, y, z)</code> Set the specified x, y, z to the <i>X3D</i> , <i>Y3D</i> , and <i>Z3D</i> properties respectively.
<code>rotate_translate(matrix)</code> Transform <i>X3D</i> , <i>Y3D</i> , <i>Z3D</i> to new coordinates according to a given rotate-translate transform matrix.
<code>translate(matrix)</code> Transform <i>X3D</i> , <i>Y3D</i> , <i>Z3D</i> to new coordinates according to a given translation transform matrix.
<code>rotate(matrix)</code> Transform <i>X3D</i> , <i>Y3D</i> , <i>Z3D</i> to new coordinates according to a given rotation transform matrix.
<code>scale(Sx, Sy, Sz)</code> Transform <i>X3D</i> , <i>Y3D</i> , <i>Z3D</i> to new coordinates according to given scaling factors. Scale the object with respect to the coordinate origin.
<code>project(viewingPoint)</code> Calculate <i>Xp</i> , <i>Yp</i> postions according to a given viewpoint position.

## Polygon

<b>Constructor Summary</b>
<code>Polygon(vertices, id, object, plane)</code> Create a polygon object with a set of specified vertices, polygon's ID, object this polygon belongs to, and a plane object representing this polygon.
<b>Property Summary</b>
<code>vertices</code> An array of Point3D objects describing vertices.
<code>id</code> A polygon ID.
<code>object</code> An object this polygon belongs to.
<code>plane</code> A plane object describing a plane equation of this polygon surface.
<code>visibility</code> A visibility property.
<b>Method Summary</b>
<code>setVisibility(value)</code> Set a given value to a visibility property.
<code>isVisible()</code> Check if this polygon is a visible polygon.
<code>getDepth()</code> get a depth of this polygon surface.

## Transform

<b>Constructor Summary</b>
<code>Transform()</code> Create a transformation matrix.
<b>Property Summary</b>
<code>matrix</code> a transform matrix.
<b>Method Summary</b>
<code>translate(Tx, Ty, Tz)</code> Create a composition transformation matrix for a translation with the specified <i>Tx</i> , <i>Ty</i> , and <i>Tz</i> .
<code>rotate(Rx, Ry, Rz)</code> Create a composition transformation matrix for a rotation with the specified angle <i>Rx</i> , <i>Ry</i> , and <i>Rz</i> . <i>Rx</i> , <i>Ry</i> , and <i>Rz</i> are the rotation angles when rotate an object about the x, y, and z axis respectively.
<code>reset()</code> Reset a transformation matrix to be an identity matrix.

## Plane

Constructor Summary
<code>Plane(p1, p2, p3)</code> Create a plane with specified successive points in a counterclockwise direction.
Property Summary
<code>A</code> A Cartesian component A from the plan equation $Ax + By + Cz + D = 0$ .
<code>B</code> A Cartesian component B from the plan equation $Ax + By + Cz + D = 0$ .
<code>C</code> A Cartesian component C from the plan equation $Ax + By + Cz + D = 0$ .
<code>visibility</code> a visibility property

## Vector

Constructor Summary
<code>Vector(A, B, C)</code> Create a vector with a specified Cartesian components.
Property Summary
<code>A</code> <i>A</i> is a value in the x-axis.
<code>B</code> <i>B</i> is a value in the y-axis.
<code>C</code> <i>C</i> is a value in the z-axis.
Method Summary
<code>size()</code> Return a size of this vector.
<code>normalize()</code> Normalize this vector to be a unit vector.
<code>addVector(a_vector)</code> Add this vector with the specified vector.
<code>dotVector(a_vector)</code> Return a scalar value of the dot product of this vector and the specified vector.

## 3. Requirements

### 3.1 Browser

The ideal browser for us to display our output document is the Mozilla-SVG browser because currently it is the only browser that natively supports SVG without installing a plug-in. While this browser is on its way to fully support SVG, at this moment, it does not support the `<linearGradient>` tag yet. We really need a `<linearGradient>` tag to perform linear shading across a polygon surface in the scene, so that we can produce realistic 3D graphics. Thus, for now, we can use only Internet Explorer to demonstrate our application.

### 3.2 DTD file

A source X3D file must be validated with the X3DToSVG.DTD file before it is sent to our translator. The translator will not check any syntax errors of a source document. Nothing will be displayed on the output page if there are syntax errors in a source file. Besides the validation of a source document, another advantage of having a DTD file is that we can define a default value to any attribute in a DTD file. The default value is automatically given to the translator through an XML parser if that attribute is not specified in the source document. This helps a lot in terms of implementation. Since it is legal for X3D tags to not have attributes, without a DTD file, we still need to check if a tag has an attribute. If it does, the translator simply takes the value of that attribute. If it does not, the translator needs to handle this situation by taking a value predefined in the

program. So, having a DTD file makes our program easier to develop and our program does not have to handle this task. (The X3DToSVG.DTD file is included in *Appendix B*.)

### 3.3 X3D Tags Supported in This Project

The table below shows the X3D tags and attributes that we support. Every attribute has its own default value defined in X3DToSVG.DTD.

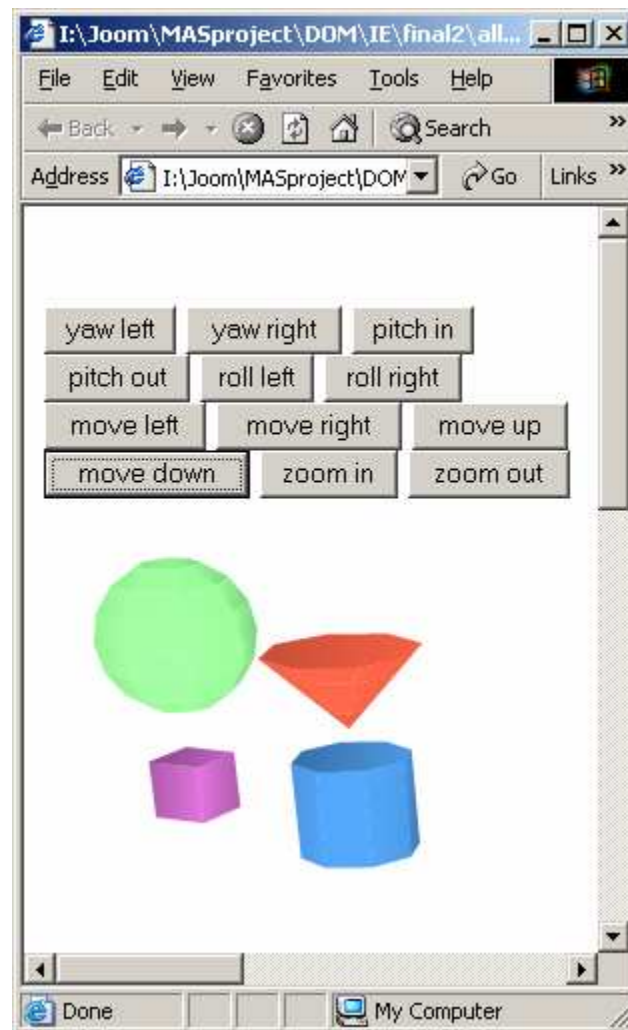
Tag	Attributes	Default attribute values
Appearance	-	
Box	size	“20 20 20”
Cone	bottomRadius height	“10” “20”
Cylinder	height radius	“20” “10”
Group	DEF USE	
Material	diffuseColor	“0.8 0.8 0.8”
Scene	-	
Shape	-	
Sphere	radius	“10”
Transform	translation rotation scale	“0 0 0” “0 0 1 0” “1 1 1”
X3D	-	

**Table 3-1** X3D tags and attributes supported by our translator.

### 3.4 Features and User Interface

There are buttons on the top of the page for users to change perspectives to view the 3D objects. Users can change perspective around the x, y, and z-axis, move left, right, up, and down, and zoom in and out. The *yaw left* and *yaw right* buttons are provided for turning perspectives around the y-axis. The *pitch in* and *pitch out* buttons are provided for turning perspectives around the x-axis. The *roll left* and *roll right* buttons are

provided for turning perspectives around the z-axis. The *move left*, *move right*, *move up*, *move down*, *zoom in*, or *zoom out* buttons are to move our viewing position from the scene in the direction of left, right, up, down, in and out respectively. These interactive features allow users to navigate a view of 3D scenes as it can be done in the X3D browser.



**Figure 3-1** Features and user interface for the user to navigate the scene.

### 3.5 The Maximum Load

As mentioned in the introduction, we developed three versions of our translator in this project. We ran several test cases on the second version, which was the best one, to find the maximum load that our translator could handle. After running many test cases on several machines having AMD AthlonXP or Pentium IV processors running at 1300 – 1400 MHz and 512 MB Memory, the program responds within one second when we change perspectives of the scene that renders up to 300-320 polygons. In other words, our program takes less than one second to produce a view of a 3D scene- whenever users change perspective- if the scene has fewer than five spheres or other combinations of objects such as fifty boxes, thirty cylinders, nine cones and one box that generate a total of fewer than three hundred polygons. Below is a table showing processing time of different test cases we ran on the second version of our translator.

# of Polygons	Processing Time (milliseconds)
32 (a cone)	~30-60
64 (a sphere)	~110-130
128 (two spheres)	~290-320
192 (three spheres)	~560-590
256 (four spheres)	~750-870
320 (five spheres)	~1090-1190
384 (six spheres)	~1590-1680
448 (seven spheres)	~1972-2133
512 (eight spheres)	~2444-2654
576 (nine spheres)	~3024-3405
640 (ten spheres)	~3225-3755
672 (ten spheres + one cone)	~3625-4266

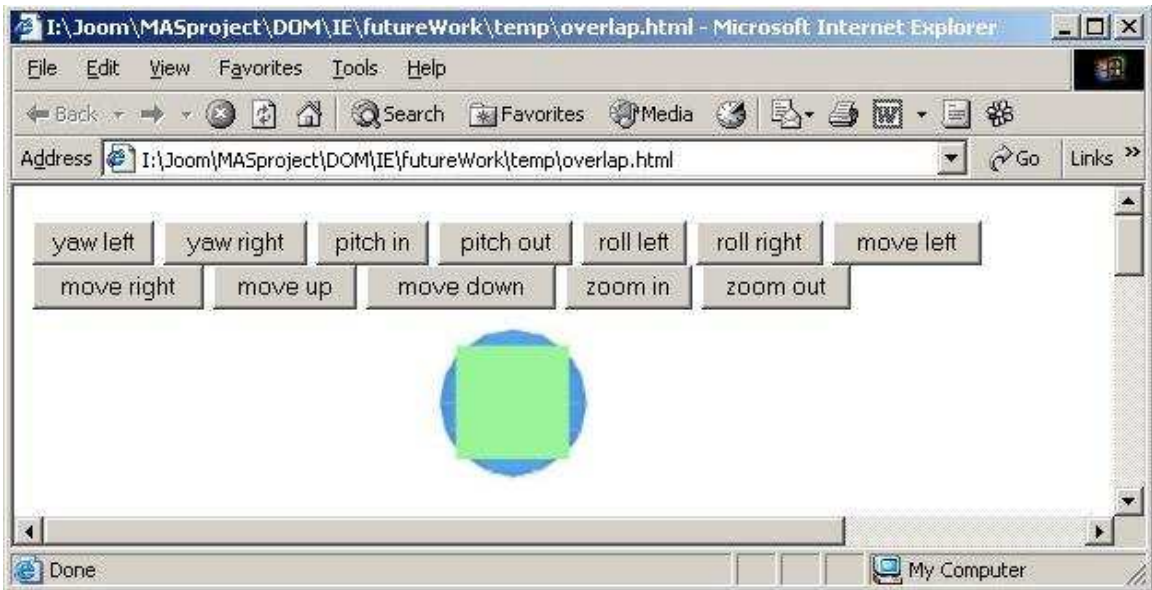
**Table 3-2** Processing time of different test cases running on the translator version 2.

We also want to compare our program's performance with other existing X3D browsers. Unfortunately, it is impossible to measure processing time consumed by the X3D browser for rendering a new scene when a perspective is changed. An X3D file is a tag-based data file and sent to be processed by X3D plugins or browsers. X3D does not provide any tag that allows us to measure time when the browser performs the rendering task. Nevertheless, it is noticeable that an existing X3D browser has a better performance than our translator when we run the same X3D document on the same machine because the X3D browser obviously renders a scene in a new perspective faster than our application does.

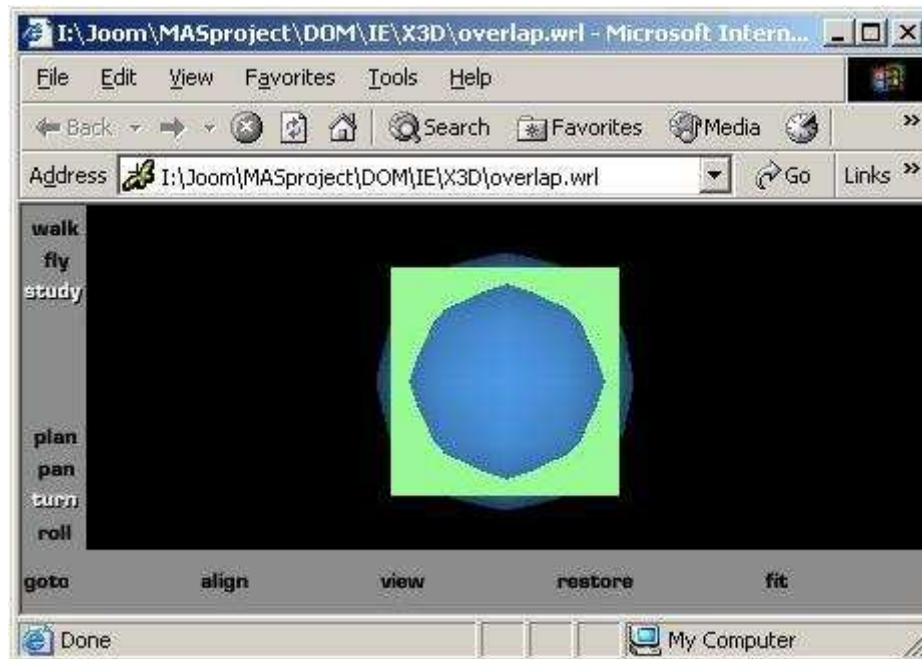
### **3.6 Limitations**

Our program cannot guarantee to display the scene of overlapping objects correctly. Figure 3-2 shows a view of an overlapping box and sphere that our application fails to display correctly. In this scenario, a box and a sphere are created at the origin, so a correct view of this 3D scene should be as displayed in the right picture; some parts of the sphere surface should appear in the middle front face of the box. However, our program fails to handle this situation. In this project - as described in section 4.14 - we render objects in depth order, that is, all of polygons of a deeper object are drawn on the screen before any of polygons of a closer object. The algorithm applied in the application draws one object a time, so our program cannot draw parts or polygons of one object over some parts or polygons of another object.





**Figure 3-2** View produced by our application.

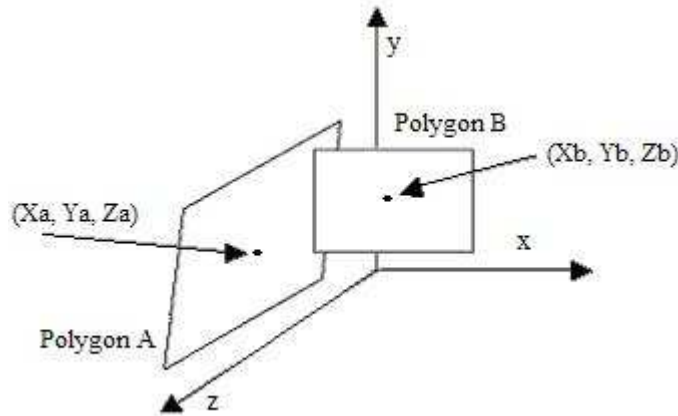


**Figure 3-3** Correct view generated by an X3D plug-in.

A possible approach to solve this problem, which has been used by several graphics applications, is to render graphics in the scene pixel by pixel. Unfortunately, this solution is not doable in SVG because SVG cannot internally describe a polygon pixel by pixel.

Although we can create a polygon as small as the size of a square pixel, this is not practical and very prohibitive to implement. Another way to handle this scenario is to subdivide a polygon so that we can display different parts of the polygon. However, breaking a polygon surface is a very complicated task and potentially has a very expensive computational cost. We decided to leave this for future work.

In our translator, we use the depth of a polygon to determine which polygon should be drawn first, second, and so on. We calculate the depth of a polygon by sampling a center point of the surface and use the z-value of this point as the depth of the polygon (more details about how to order polygons and sample a center point are in section 4.14). In the scenario that two polygons are intersecting each other as displayed in the picture below, our program cannot produce a correct view of the scene.



**Figure 3-4** Correct view of two polygons intersecting each other.

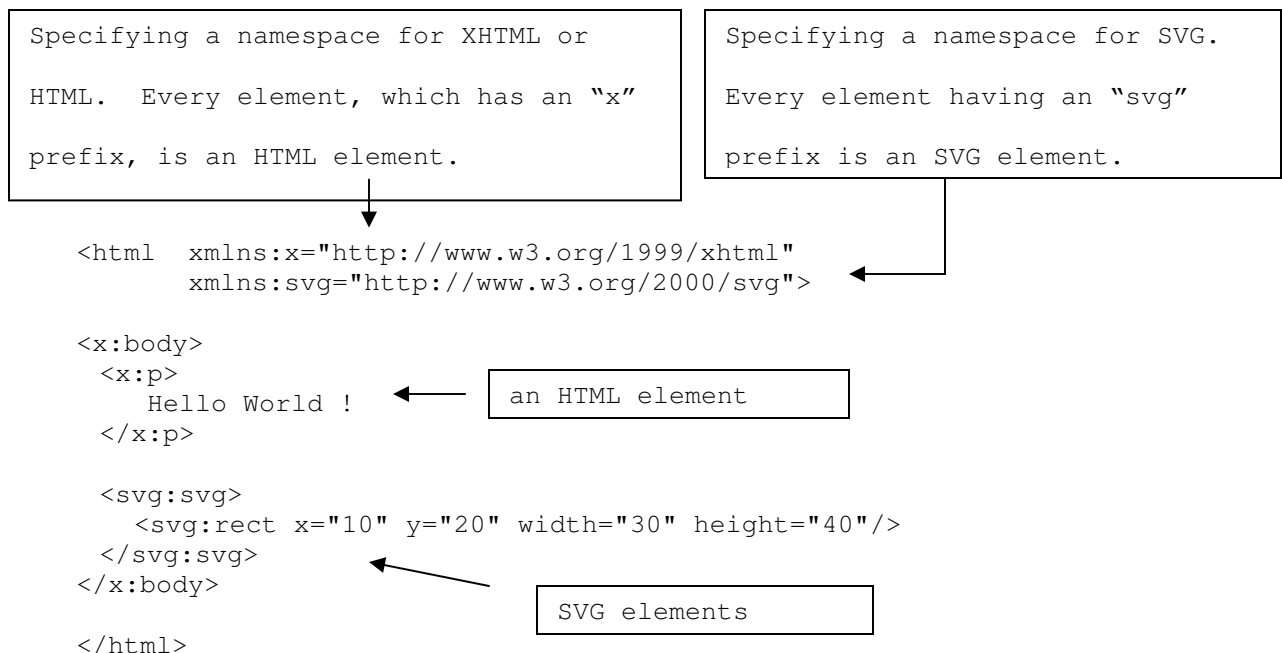
Suppose the viewing position is on the positive z-axis far away from the origin. By using the z-value of the middle point of a polygon as depth, polygon A is closer to the viewpoint than polygon B. Hence, polygon B will be rendered before polygon A.

However, in order to generate a correct view as shown above, polygon B should be drawn after polygon A. The estimation of a polygon's depth by using a z-value of a center point cannot handle this situation. To display a view of this scene correctly, we have to subdivide polygons where the intersection occurs. At this time, we have not implemented a process of subdividing polygons. We leave this for future work. Even though the algorithm we use right now in our project is not perfect and cannot handle some cases, this method is fast and simple, and requires a small number of calculations.

## 4. Implementation

### 4.1 The Output Document's Type

The output document in this project, which renders a view of 3D objects in SVG format, is actually a mixture of two types of documents, dynamic HTML and SVG. This kind of a mixed document can be considered as an XHTML (eXtensible HyperText Markup Language) document. Both Internet Explorer and an SVG-enabled Mozilla browser support the mixed SVG and DHTML document and display data correctly. These browsers can differentiate which elements belong to which language by declaring a namespace prefix. Below is a sample of a document mixing the two languages: HTML and SVG.



**Figure 4-1** Sample of an XHTML document with both HTML and SVG.

In this project, we apply this technique to embed SVG tags into an HTML document to display SVG data as a view of 3D scenes.

#### 4.2 Access/Load A Source X3D File

As mentioned earlier in the introduction section, one of the two processes in our translator is a translation process. The first thing we need to do in a translation process is to access a source X3D file. We use the `document.load()` command for loading and reading data from an X3D document. Below is a code sample for loading an X3D document from the specified URL in IE, Mozilla, and Netscape.

```
A Mozilla and Netscape solution.  
// Create a new Document object.  
var x3dDoc = document.implementation.createDocument("", "", null);  
// Tell it what URL to load.  
x3dDoc.load(url);
```

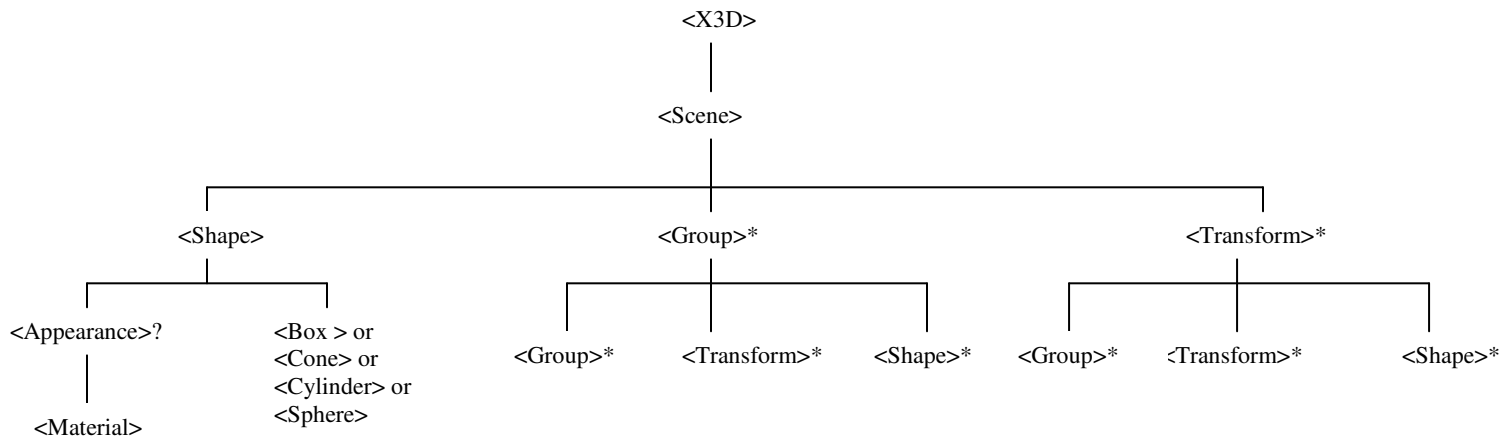
```
An IE solution.  
// Use Microsoft's proprietary API for IE to create a document.  
var x3dDoc = ActiveXObject("Microsoft.XMLDOM");  
// Tell it what URL to load.  
x3dDoc.load(url);
```

A sample of code for loading an X3D file in IE, Mozilla, and Netscape.

The implementation is slightly different between loading a file in IE and loading a file in Mozilla and Netscape. After executing the above sample, the *x3dDoc* variable contains a parse structure of the input file. We will use this *x3dDoc* later to determine which primitives will be created and where they should be rendered in the scene.

### 4.3 A DOM Structure of An X3D File

To efficiently handle all X3D tags supported by this translator as mentioned in section 3, and produce a view of 3D scenes correctly, we modify the parse structure of a source X3D file so that the process of handling the supported tags and performing a translation task is easy to implement. In this project, we did not directly traverse the parse structure of an input file but modified the original DOM structure to take out data for generating 3D objects. Before saying how these modifications were made, we need to understand the structure of an X3D document. Figure 4-2 shows the tree structure of X3D tags we support in this project.



**Figure 4-2** Tree structure of X3D tags we support in this project.

An asterisk in the above picture means it can have zero or more of these tags. For example, the `<Group>` tag can have zero or more of `<Shape>` tags as child nodes. A question mark means this tag is optional. For instance, the `<Shape>` tag can either have zero or one `<Appearance>` tag.

The <X3D> tag is a root element of an X3D document and has only one <Scene> element as child node. The <Scene> tag can have zero or more <Shape>, <Group>, and <Transform> tags as child nodes and so can <Group> and <Transform> tags. The <Box>, <Cone>, <Cylinder>, and <Sphere> tags are the primitive tags to create a box, cone, cylinder, and sphere respectively. The <Shape> tag must have only one of these primitive tags as child node and optionally has the <Appearance> tag to specify a color of the primitive. The <Transform> tag is to specify a translation, rotation, and scale transformation. We can reposition, scale, and rotate objects through this tag. Both <Group> and <Transform> tags have a recursive structure. Usually, an X3D document is organized by many nested structures of these tags to create complicated objects. A sample of using these tags to create a complicated object is shown below.

```

<X3D>
  <Scene>

    <Group DEF="golf">

      <Transform translation="0 10 0">
        <Shape>
          <Cone bottomRadius="15"
                height="5"/>
        </Shape>
      </Transform>

      <Transform translation="0 20 0">
        <Shape>
          <Box size="2 10 2"/>
        </Shape>
      </Transform>

      <Transform>
        <Shape>
          <Appearance>
            <Material
              diffuseColor="0.9 0.9 0.9"/>
          </Appearance>
          <Sphere radius="10"/>
        </Shape>
      </Transform>

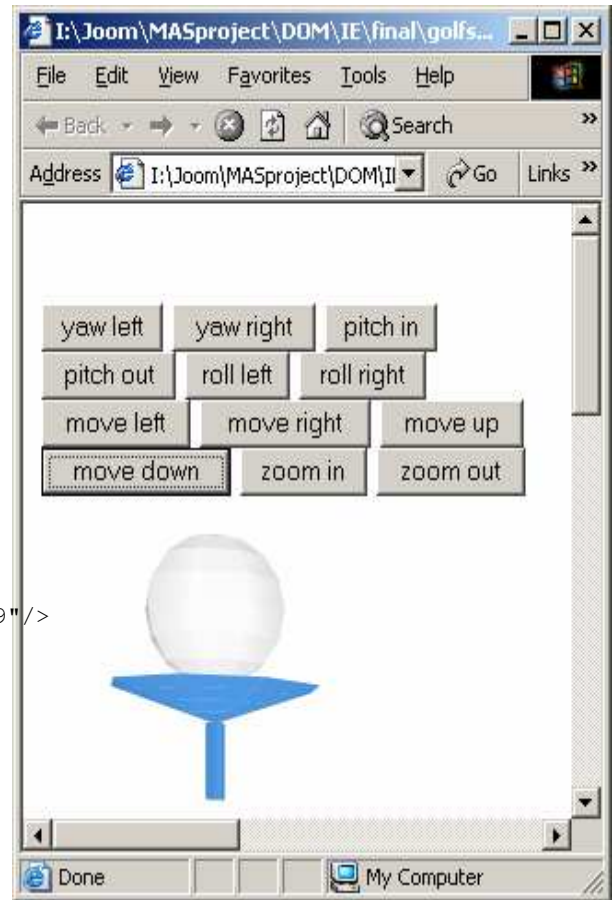
    </Group>

  </Scene>
</X3D>

```

A tee

A golf ball



**Figure 4-3** A sample X3D document.

One of the other tags that we support is the `<Group>` tag. To support this tag, we have to modify the original parse structure of a source document before taking data out of it as explained later. As its name implies, this tag's purpose is to group its child nodes together. However, the most powerful use of this tag is not grouping the child nodes together; it is to avoid repeatedly describing a group of the same objects by calling the group's name. Below is a sample showing how to use the `<Group>` tag to create three golf balls with a tee. Instead of redundantly describing a golf ball sitting on a tee a few



more times, after describing a group of a golf ball with a tee, we can re-use this group by calling its name to re-describe the same objects.

```

<X3D>
  <Scene>
    <Group DEF="golf">
      <Transform translation="0 10 0">
        <Shape>
          <Cone bottomRadius="15" height="5"/>
        </Shape>
      </Transform>

      <Transform translation="0 20 0">
        <Shape>
          <Box size="2 10 2"/>
        </Shape>
      </Transform>

      <Transform>
        <Shape>
          <Appearance>
            <Material diffuseColor="0.9 0.9 0.9"/>
          </Appearance>
          <Sphere radius="10"/>
        </Shape>
      </Transform>
    </Group>

    <Transform translation="50 0 0">
      <Group USE="golf"/>
    </Transform>

    <Transform translation="-50 0 0">
      <Group USE="golf"/>
    </Transform>
  </Scene>
</X3D>

```

This tag is equivalent to a group tag named "golf" along with its descendent.

**Figure 4-4** A sample of using <Group> tags to draw the same objects.

The *DEF* attribute of the <Group> tag is to specify a name of a group and the *USE* attribute is to put the name of a group we want to call.

#### 4.4 Manipulate A DOM Structure of A Source X3D File

The `<Shape>` node is the most important node in terms of implementation. By tracking down its child nodes, we know the information about what primitives will be created. By keeping track of its nested `<Transform>` parent nodes, we know information about translation, rotation, and scale transformations of these primitives. When we traverse the DOM structure of an input X3D file, we create a 3D object when we hit a `<Shape>` node. However, while traversing along the same structure, how can we create an object when we hit a `<Group>` tag that calls another `<Group>` tag? The easiest way to handle this is to replace a calling `<Group>` tag with its associated group. Hence, we want to modify the original tree structure by replacing all calling `<Group>` tags with their associated group along with the descendant nodes before taking out data to render 3D objects. Unfortunately, DOM levels I and II do not allow us to add new nodes into the DOM structure of an external file. As a result, we cannot replace a calling `<Group>` tag with a copy of its associated `<Group>` tag. We have to create a new function for cloning the original DOM structure, which is considered to be elements of an output document, so that we can arbitrarily manipulate a new cloned DOM structure. In this project, we use this strategy to construct 3D objects on the screen. After substituting every calling `<Group>` node with its associated `<Group>` node in a new cloned structure, we use a DOM command `document.getElementsByTagName()` to get all `<Shape>` nodes from the modified cloned structure and keep them in an array. Then we process them one by one. We create a primitive from each `<Shape>` node and use the bottom-up approach to get transformation data through its nested `<Transform>`

parent nodes. The code below shows how we use the DOM API to modify and clone a DOM structure of a source file.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:svg="http://www.w3.org/2000/svg">
  /*
  * buildSVG( ) is executed when the <body> node gets loaded. This
  * function will create SVG nodes for displaying 3D objects and
  * insert the new SVG nodes to the output document as child
  * nodes of the <body> tag.
  */
  <body id="start" onload="buildSVG()">
    <p>
      /@
      Create buttons for changing perspectives here.
      <input type="button" value="yaw left" onclick="yawLeft( )"/>
      @/
    </p>
  </body>

  <script type="text/javascript">
    /*
    * Use the DOM 2 technique, if it is supported (for Mozilla
    * and Netscape)
    */
    if(document.implementation &&
       document.implementation.createDocument)
    {
      var x3dDoc =
        document.implementation.createDocument("", "", null);
      /*
      * This is to assure that the parser halts the
      * execution until the source file is fully loaded.
      */
      x3dDoc.async = "false";
      x3dDoc.load("allPrimitives.xml");
    }
    /* Otherwise, use Microsoft proprietary API for IE */
    else if(window.ActiveXObject)
    {
      var x3dDoc = new ActiveXObject("Microsoft.XMLDOM");
      x3dDoc.async = "false";
      /* Load a source X3D file. */
      x3dDoc.load("allPrimitives.xml");
    }
    /*
    * Get the <Scene> node from the parse structure of an
    * input file.
    */
    var nodes = x3dDoc.getElementsByTagName("Scene");
    var scene = nodes[0];
    /*
```

```

* Clone g_scenel from scene. g_scenel is an element of
* this document while scene is an element of the source
* document. Both of them have the same structure of their
* descendent.
*/
var g_scenel = cloneElement(scene);
//This is to replace calling groups with associated groups.
modifyX3D( )
// Create an SVG root node.
var g_svg = svg_root_builder();
/*
* Get an HTML <body> node of this document where we want to
* insert SVG tags.
*/
var g_bodyNode = document.getElementById("start");
function buildSVG( )
{
    /*
    * Get the entire <Shape> nodes from the modified DOM
    * structure.
    */
    var shapeNodes = g_scenel.getElementsByTagName("Shape");
    var len = shapeNodes.length;
    for(var i=0; i<len; i++)
    {
        /*
        Create SVG elements for each object according to
        shapeNodes and append them to an SVG root node.
        */
    }
    /*
    * Insert an SVG root into the output document as
    * child of the body node.
    */
    g_bodyNode.appendChild(g_svg);

    // Render each object in the scene.
    drawObjects( );
}
</script>
</html>

```

The output document is a mixture of HTML and SVG. We need to declare a namespace for SVG elements in the <html> tag; in this case, we use “svg” as prefix for an SVG element. Since the Mozilla-SVG browser does not yet support displaying shading on a polygon surface, to view shading we have to use IE to display the output document at this time. When the output document is loaded (<body> is on loaded), buildSVG( ) will

be executed immediately. We create the buttons for changing perspective between `<p>` and `</p>`. Each button will call an event-handling function to generate a new view. However, a few things need to be done before `buildSVG()` gets processed. First, we need to load a source X3D file by the `document.load()` command. Then we want to get a `<Scene>` node from the DOM structure because it has all other nodes we need for our translation task. We get this node by using a `document.getElementsByTagName()` command and clone this node by calling `cloneElement(...)`, which returns a new cloned node with its descendant as element of the output document. Then we modify a cloned `<Scene>` node by calling `modifyX3D()`, which replaces every calling `<Group>` node in this structure with another related `<Group>` node and its descendants. The last thing that needs to be done before `buildSVG()` is executed is to create a root SVG node, which is done by `svg_root_builder()`. `buildSVG()` will get the entire `<Shape>` nodes from a new cloned modified structure, create SVG elements for each object according to a `<Shape>` node, and append them to the SVG root node. After dynamically creating related SVG elements for all objects in the scene and appending them to the root node. Lastly `buildSVG()` inserts the root node with its descendants into this output document as a child node of an HTML `<body>` node.

```
// Clone a new node.
function cloneElement(node)
{
    var attr = node.attributes;
    var len = attr.length;
    var name = node.nodeName;
    var element = document.createElement(name);
    var attributes = new Array(len);
```

```

for(var i=0; i<len; i++)
{
    attributes[i] = document.createAttribute(attr[i].name);
    attributes[i].value = attr[i].value;
    element.attributes.setNamedItem(attributes[i]);
}
// Recursively clone child nodes of the prototype node.
if(node.hasChildNodes())
{
    var children = node.childNodes;
    var len1 = children.length;
    var cloneChild = new Array(len1);
    for(var i=0; i<len1; i++)
    {
        cloneChild[i] = cloneElement(children[i]);
        element.appendChild(cloneChild[i]);
    }
    return element;
}
else
    return element;
}

```

cloneElement(node) is used to clone a new element of the output document. It has one argument given by the caller. It returns a new element that has the same structure as an argument node. The argument node and its descendant are recursively cloned by this function as shown in the code above. The new cloned element is created by a document.createElement() command and other attributes associated with this element are created by a document.createAttribute() and document.setNameItem() command.

```

// Replace every calling group with its associated group.
function modifyX3D( )
{
    // Get the entire <Group> nodes from the cloned DOM structure.
    var groupNodes = g_scenel.getElementsByTagName("Group");

    var groupDEF = new Array();
    var countDEF = 0;
    var groupUSE = new Array();
    var countUSE = 0;
    /*
    * Separate group nodes into two categories, groupDEF and

```

```

* groupUSE.
*/
for(var i=0; i<groupNodes.length; i++)
{
    if(groupNodes[i].getAttribute("DEF") != null)
        groupDEF[countDEF++] = groupNodes[i];
    else if(groupNodes[i].getAttribute("USE") != null)
        groupUSE[countUSE++] = groupNodes[i];
}
for(var i=0; i<groupUSE.length; i++)
{
    var targetName = groupUSE[i].getAttribute("USE");
    /*
    * Search a group whose DEF value is targetName from the
    * groupDEF array.
    */
    var targetGroup = getNodeFromGroup(groupDEF, targetName);
    var dup = targetGroup.cloneNode(true);
    var parent = groupUSE[i].parentNode;
    // Replace this node of the groupUSE with the targetGroup.
    parent.replaceChild(dup, groupUSE[i]);
}
}

```

`modifyX3D()` is used to replace every calling `<Group>` with another associated `<Group>` in the structure. First, we need to get the entire `<Group>` nodes from the cloned structure, and then separate `<Group>` nodes into two groups, `groupDEF` and `groupUSE`. A `<Group>` node that has the *DEF* attribute will be in `groupDEF`, and a `<Group>` node that has the *USE* attributes will be in `groupUSE`. Every node in `groupUSE` will be replaced with an associated node from `groupDEF`.

```

// This function uses DOM methods to create an SVG root and other
related attributes.
function svg_root_builder( )
{
    // Create an "svg" element.
    var svg = document.createElement("svg:svg");

    // Create attributes of the "g_svg" element.
    var width = document.createAttribute("width");
    var height = document.createAttribute("height");
    var viewBox = document.createAttribute("viewBox");

    // Assign default value to a width, height and viewBox attribute.
    width.value = "20cm";

```

```

    height.value = "20cm";
    viewBox.value = "-100 -50 150 200";

    // Bind the attributes to the element.
    svg.attributes.setNamedItem(width);
    svg.attributes.setNamedItem(height);
    svg.attributes.setNamedItem(viewBox);

    return svg;
}

```

`svg_root_builder()` uses a `document.createElement()` command to create the SVG root node and a `document.createAttribute()` and `document.setNameItem()` command to create other attributes of this node.

#### **4.5 Using DOM API to Dynamically Create and Insert SVG Tags to The Output Document.**

As mentioned before, we use `<polygon>` tags to create a mesh of polygons to form 3D objects and `<linearGradient>` tags to apply a linear shading pattern across polygon surfaces. These tags are dynamically created and added into the output document by DOM. Before we explain how we implement this, we want to show the structure of these two SVG tags first.

A `<linearGradient>` tag must be created as a child node of a `<defs>` tag. The `<defs>` tags are used to define tags that we do not want to display on the screen. This tag has an ID attribute for specifying a name to this tag, so that it can be referred to later by its ID. A `<linearGradient>` tag has `<stop>` as child nodes for specifying a starting and stop color.



```

<defs>
<linearGradient id="test">
  <stop offset="0%" stop-color="rgb(125, 125, 125)"/>
  <stop offset="100%" stop-color="rgb(130, 130, 130)"/>
</linearGradient>
</defs>
<polygon points="10,30 20,25 45,20" fill="url(#test)" stroke="none"
stroke-width="0.4"/>

```

The first `<stop>` child node is used to specify a starting color and the second `<stop>` child node is used to specify a stopping color. A `<stop>` tag requires two attributes; the `offset` tells the point along the line from one side to another side of a polygon at which the color should be equal to the `stop-color`. The offset is expressed as a percentage from 0 to 100%. The `stop-color` is in the form `rgb(red-value, green-value, blue-value)`, where each value is in the range 0-255.

A `<polygon>` tag allows us to specify a series of *points* that describe a geometric area. The point attribute consists of a series of x- and y- coordinate pairs separated by comma or white space. We do not have to return to the starting point; the shape will be automatically closed. We can fill the polygon with a color or a shading pattern through the `fill` attribute. In the above sample, we want to apply a linear gradient to this polygon, so in the `fill` attribute, we refer to the `<linearGradient>` tag by its name in the format `url(#tagID)`. We can specify a stroke color in the `stroke` attribute and a stroke width in the `stroke-width` attribute.

In this application, we create five core classes – one superclass, `Primitive`, and four subclasses, `Box`, `Cone`, `Cylinder`, and `Sphere` - to perform the translation and rendering

task to generate a view of a 3D scene. Superclass Primitive has general properties for its four subclasses. Each subclass has specific details about how to model its object. Both `<linearGradient>` and `<polygon>` tags are dynamically created and appended to the SVG root node by the `Primitive.createTag()` method. The SVG root node is inserted into the document later by `buildSVG()`. Next we show the code of this method along with comments on how to use the DOM API to dynamically create these two tags. Basically, we use the `document.createElement()` command to create a new element for this document, and `document.createAttribute()` command to create a new attribute. After that, we need to bind the new attribute to the new element by using the `Element.attributes.setNamedItem()` command.

```
Primitive.prototype.createTag = function()
{
    /*
    * Create a "defs" element where <linearGradient> tags will be
    * inserted as children nodes.
    */
    var defs = document.createElement("svg:defs");
    /* Append a <defs> tag into the <svg> root node. */
    svg.appendChild(defs);
    /*
    * Create <polygon> and <linearGradient> tags. The number of
    * <polygon> and <linearGradient> created is equal to the number
    * of polygons we use to form the primitive; for example, a sphere
    * is built up of 64 polygons, so 64 <polygon> tags and 64
    * <linearGradient> tags are created.
    */
    for(var k=0; k<this.num_faces; k++)
    {
        /* Create a "polygon" element. */
        polygons[k] = document.createElement("svg:polygon");
        /*
        * Create attributes of the "polygon" element and assign
        * values to each attribute in such a way that the polygon
        * appears invisible.
        */
        points[k] = document.createAttribute("points");
        points[k].value = "0, 0";
        fill[k] = document.createAttribute("fill");
        fill[k].value = "none";
    }
}
```

```

stroke[k] = document.createAttribute("stroke");
stroke[k].value = "none";
// Bind polygon's attribute nodes to the "polygon" element
polygons[k].attributes.setNamedItem(points[k]);
polygons[k].attributes.setNamedItem(fill[k]);
polygons[k].attributes.setNamedItem(stroke[k]);

/*
 * Store a <polygon> tag as global variable, which will be
 * called later to draw the polygon.
 */
polygonSVGTag[countTag++] = polygons[k];
/* Append a <polygon> tag into the <svg> root node. */
svg.appendChild(polygons[k]);

/* Create a "linearGradient" element for shading. */
linearGradient[k] =
    document.createElement("svg:linearGradient");
/*
 * Create an id attribute of a "linearGradient" element. Use
 * an object's id and polygon's id as a unique of this tag.
 */
id[k] = document.createAttribute("id");
id[k].value = "" + this.id + k;
//Bind an "id" attribute to the "linearGradient" element.
linearGradient[k].attributes.setNamedItem(id[k]);
/* Append a <linearGradient> tag to a <defs> tag. */
defs.appendChild(linearGradient[k]);

/*
 * Create a "stop" element as well as an "offset" attribute
 * where we can specify staring and stopping colors on the
 * left and right edge of the polygon surface for shading.
 */
stopL[k] = document.createElement("svg:stop");
offsetL[k] = document.createAttribute("offset");
offsetL[k].value = "0%";
stop_colorL[k] = document.createAttribute("stop-color");

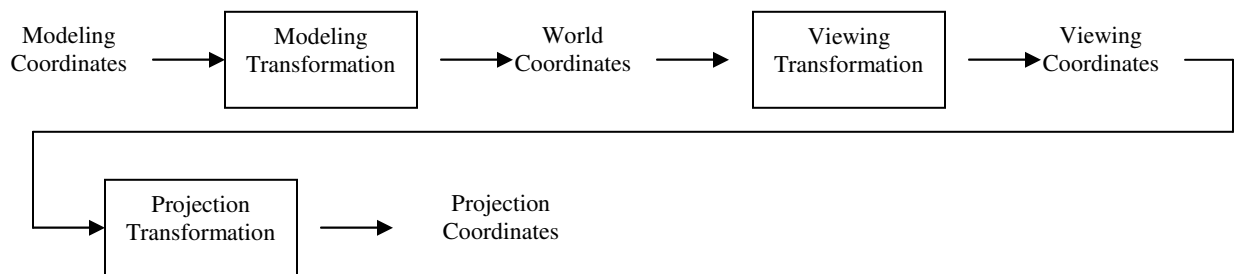
/* Bind stop's attribute nodes to the "stop" element */
stopL[k].attributes.setNamedItem(offsetL[k]);
stopL[k].attributes.setNamedItem(stop_colorL[k]);
stopR[k] = document.createElement("svg:stop");
offsetR[k] = document.createAttribute("offset");
offsetR[k].value = "100%";
stop_colorR[k] = document.createAttribute("stop-color");
stopR[k].attributes.setNamedItem(offsetR[k]);
stopR[k].attributes.setNamedItem(stop_colorR[k]);

/* Append a <stop> tag to a <linearGradient> tag. */
linearGradient[k].appendChild(stopL[k]);
linearGradient[k].appendChild(stopR[k]);
}
}

```

## 4.6 Viewing Pipeline Technique

This mechanism is used to generate a view of 3D scenes on a computer screen. The first step is to model 3D objects in world coordinates. Once the object has been modeled, world coordinate positions are converted to viewing coordinates. The viewing-coordinate system is used as a reference for specifying the observer's viewing positions and the position of the projection plane which, in this case, we can think of as the computer screen. The viewing-coordinate description changes, if the user changes perspectives. After we get viewing-coordinate positions, projection operations are performed to convert the viewing-coordinate description to projected positions on the projection plane. These will then be mapped to the computer screen. If the projected positions are outside the viewing limits, we can clip the object. However, in this project, we do not perform clipping operations and leave them for future work. The parts outside the viewing space are still rendered by our translator.



**Figure 4-5** General three-dimensional transformation pipeline.

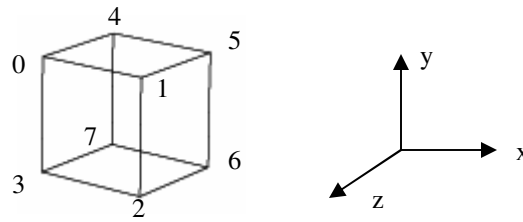
The above picture shows the general processing steps for modeling and converting a world-coordinate description of a scene to projection coordinates.

## 4.7 3D Object Models

The first step in the general three-dimensional transformation pipeline is performing a modeling transformation to get a world-coordinate description describing a box, a cone, a cylinder, and a sphere.

### Modeling a box

A box modeled as a mesh of six polygons with a center at the origin is illustrated below.



**Figure 4-6** Box model.

The X3D <Box> tag has a “size” attribute, which specifies the dimension of a box. We model x-y-z coordinates of each polygon vertex from the width, height, and depth we got from the “size” attribute. These x-y-z coordinates are world coordinates. Let  $x$  be a width,  $y$  be a height, and  $z$  be a depth. The following are the coordinates of the vertices we use.

$v0$  (world coordinates at vertex 0) =  $(-x/2, y/2, z/2)$

$v1 = (x/2, y/2, z/2)$ ,  $v2 = (x/2, -y/2, z/2)$ ,  $v3 = (-x/2, -y/2, z/2)$ ,  $v4 = (-x/2, y/2, -z/2)$

$v5 = (x/2, y/2, -z/2)$ ,  $v6 = (x/2, -y/2, -z/2)$ ,  $v7 = (-x/2, -y/2, -z/2)$

Each polygon of a box has been modeled with a set of specific vertices as shown below.

$\text{polygon0} = [v0, v3, v2, v1]$ ,  $\text{polygon1} = [v5, v6, v7, v4]$ ,

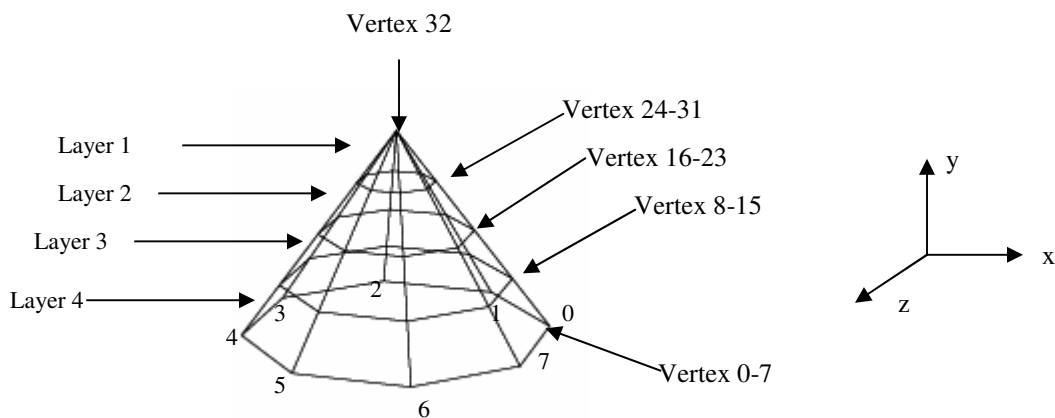
$\text{polygon2} = [v4, v7, v3, v0]$ ,  $\text{polygon3} = [v1, v2, v6, v5]$ ,

polygon4 = [v4, v0, v1, v5],    polygon5 = [v3, v7, v6, v2]

Notice, we model a set of vertices for each polygon in a counter-clockwise order viewed from the outside because it will be useful later when we want to find a normal vector representing this polygon.

### Modeling a cone

A cone is modeled as a mesh of thirty-three polygons and thirty-three vertices; thirty-two polygons are side surfaces and one is the bottom. A cone is created with the bottom lying down on the z-x plane.



**Figure 4-7** Cone model.

The X3D <Cone> node provides information about the radius and height from its “bottomRadius” and “height” attributes. We use this information to model a world coordinate of each polygon vertex. A cone that we model in this project is composed of four layers of polygons. Each layer has eight polygons. Notice, polygons in the top layer are triangles but the others are rectangles. Here  $r$  is the cone’s radius and  $h$  is the cone’s height. Below is a table of how to calculate the coordinates of each vertex.

Vertex	x coordinate	y coordinate	Z coordinate
v0	$r$	$0$	$0$
v1	$r \times \cos(\pi/4)$	$0$	$-r \times \sin(\pi/4)$
v2	$r \times \cos(2\pi/4)$	$0$	$-r \times \sin(2\pi/4)$
v3	$r \times \cos(3\pi/4)$	$0$	$-r \times \sin(3\pi/4)$
v4	$r \times \cos(4\pi/4)$	$0$	$-r \times \sin(4\pi/4)$
v5	$r \times \cos(5\pi/4)$	$0$	$-r \times \sin(5\pi/4)$
v6	$r \times \cos(6\pi/4)$	$0$	$-r \times \sin(6\pi/4)$
v7	$r \times \cos(7\pi/4)$	$0$	$-r \times \sin(7\pi/4)$
v8	$3r/4$	$h/4$	$0$
v9	$3r/4 \times \cos(\pi/4)$	$h/4$	$-3r/4 \times \sin(\pi/4)$
v10	$3r/4 \times \cos(2\pi/4)$	$h/4$	$-3r/4 \times \sin(2\pi/4)$
v11	$3r/4 \times \cos(3\pi/4)$	$h/4$	$-3r/4 \times \sin(3\pi/4)$
v12	$3r/4 \times \cos(4\pi/4)$	$h/4$	$-3r/4 \times \sin(4\pi/4)$
v13	$3r/4 \times \cos(5\pi/4)$	$h/4$	$-3r/4 \times \sin(5\pi/4)$
v14	$3r/4 \times \cos(6\pi/4)$	$h/4$	$-3r/4 \times \sin(6\pi/4)$
v15	$3r/4 \times \cos(7\pi/4)$	$h/4$	$-3r/4 \times \sin(7\pi/4)$
v16	$2r/4$	$2h/4$	$0$
v17	$2r/4 \times \cos(\pi/4)$	$2h/4$	$-2r/4 \times \sin(\pi/4)$
v18	$2r/4 \times \cos(2\pi/4)$	$2h/4$	$-2r/4 \times \sin(2\pi/4)$
v19	$2r/4 \times \cos(3\pi/4)$	$2h/4$	$-2r/4 \times \sin(3\pi/4)$
v20	$2r/4 \times \cos(4\pi/4)$	$2h/4$	$-2r/4 \times \sin(4\pi/4)$
v21	$2r/4 \times \cos(5\pi/4)$	$2h/4$	$-2r/4 \times \sin(5\pi/4)$
v22	$2r/4 \times \cos(6\pi/4)$	$2h/4$	$-2r/4 \times \sin(6\pi/4)$
v23	$2r/4 \times \cos(7\pi/4)$	$2h/4$	$-2r/4 \times \sin(7\pi/4)$
v24	$r/4$	$3h/4$	$0$
v25	$r/4 \times \cos(\pi/4)$	$3h/4$	$-r/4 \times \sin(\pi/4)$
v26	$r/4 \times \cos(2\pi/4)$	$3h/4$	$-r/4 \times \sin(2\pi/4)$
v27	$r/4 \times \cos(3\pi/4)$	$3h/4$	$-r/4 \times \sin(3\pi/4)$
v28	$r/4 \times \cos(4\pi/4)$	$3h/4$	$-r/4 \times \sin(4\pi/4)$
v29	$r/4 \times \cos(5\pi/4)$	$3h/4$	$-r/4 \times \sin(5\pi/4)$
v30	$r/4 \times \cos(6\pi/4)$	$3h/4$	$-r/4 \times \sin(6\pi/4)$
v31	$r/4 \times \cos(7\pi/4)$	$3h/4$	$-r/4 \times \sin(7\pi/4)$
v32	$0$	$h/4$	$0$

**Table 4-1** World coordinates of a cone model.

Each polygon used to build a cone is made up of specific vertices. Below is a table of how we assign a set of specific vertices to a polygon.

Polygon	A set of vertices arranged in counter-clockwise order
polygon0	[v0, v1, v9, v8]
polygon1	[v1, v2, v10, v9]
polygon 2	[v2, v3, v11, v10]
polygon 3	[v3, v4, v12, v11]

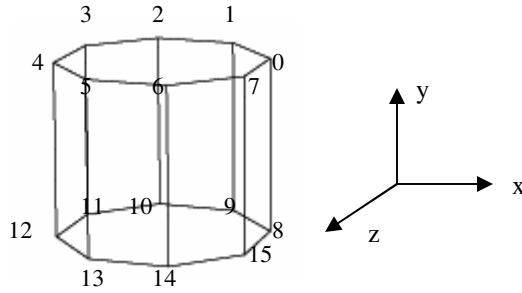
polygon 4	[v4, v5, v13, v12]
polygon 5	[v5, v6, v14, v13]
polygon 6	[v6, v7, v15, v14]
polygon 7	[v7, v0, v8, v15]
polygon 8	[v8, v9, v17, v16]
polygon 9	[v9, v10, v18, v17]
polygon 10	[v10, v11, v19, v18]
polygon 11	[v11, v12, v20, v19]
polygon 12	[v12, v13, v21, v20]
polygon 13	[v13, v14, v22, v21]
polygon 14	[v14, v15, v23, v22]
polygon 15	[v15, v8, v16, v23]
polygon 16	[v16, v17, v25, v24]
polygon 17	[v17, v18, v26, v25]
polygon 18	[v18, v19, v27, v26]
polygon 19	[v19, v20, v28, v27]
polygon 20	[v20, v21, v29, v28]]
polygon 21	[v21, v22, v30, v29]]
polygon 22	[v22, v23, v31, v30]]
polygon 23	[v23, v16, v24, v31]
polygon 24	[v24, v25, v32]]
polygon 25	[v25, v26, v32]
polygon 26	[v26, v27, v32]
polygon 27	[v27, v28, v32]
polygon 28	[v28, v29, v32]
polygon 29	[v29, v30, v32]
polygon 30	[v30, v31, v32]
polygon 31	[v31, v24, v32]
polygon 32	[v7, v6, v5, v4, v3, v2, v1, v0]

**Table 4-2** Polygon surfaces with their assigned vertices of a cone model .



### Modeling a cylinder

A cylinder is rendered as a mesh of ten polygons with sixteen vertices; eight polygons are side surfaces, two polygons are the top and bottom. It is created with its center at the origin.



**Figure 4-8** Cylinder model.

The X3D <Cylinder> node provides information about the radius and height from its “radius” and “height” attributes. We use this information to model a world coordinate of each polygon vertex. Here  $r$  is the cylinder’s radius and  $h$  is the cylinder’s height. The following are the coordinates of the vertices we use.

$$v0 = (r, h/2, 0), \quad v1 = (r \times \cos(\pi/4), h/2, -r \times \sin(\pi/4)),$$

$$v2 = (0, h/2, -r), \quad v3 = (-r \times \cos(\pi/4), h/2, -r \times \sin(\pi/4)),$$

$$v4 = (-r, h/2, 0), \quad v5 = (-r \times \cos(\pi/4), h/2, r \times \sin(\pi/4)),$$

$$v6 = (0, h/2, r), \quad v7 = (r \times \cos(\pi/4), h/2, r \times \sin(\pi/4)),$$

$$v8 = (r, -h/2, 0), \quad v9 = (r \times \cos(\pi/4), -h/2, -r \times \sin(\pi/4)),$$

$$v10 = (0, -h/2, -r), \quad v11 = (-r \times \cos(\pi/4), -h/2, -r \times \sin(\pi/4)),$$

$$v12 = (-r, -h/2, 0), \quad v13 = (-r \times \cos(\pi/4), -h/2, r \times \sin(\pi/4)),$$

$$v14 = (0, -h/2, r), \quad v15 = (r \times \cos(\pi/4), -h/2, r \times \sin(\pi/4))$$

Each polygon in a cylinder is made up of specific vertices as shown below.

polygon0 = [v0, v8, v9, v1],    polygon1 = [v1, v9, v10, v2],

polygon2 = [v2, v10, v11, v3], polygon3 = [v3, v11, v12, v4],

polygon4 = [v4, v12, v13, v5], polygon5 = [v5, v13, v14, v6],

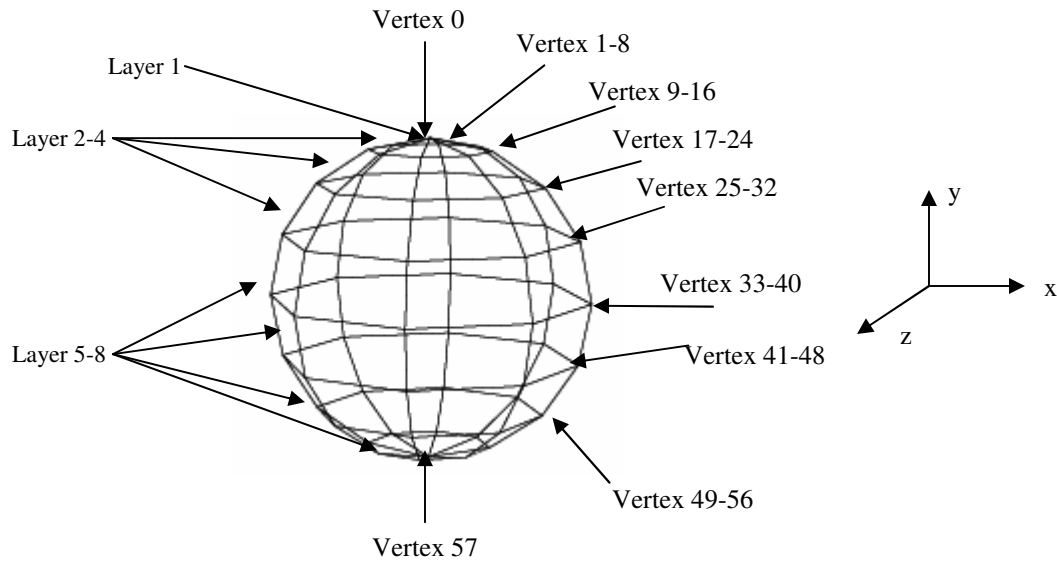
polygon6 = [v6, v14, v15, v7], polygon7 = [v7, v15, v8, v0],

polygon8 (top) = [v0, v1, v2, v3, v4, v5, v6, v7],

polygon9 (bottom) = [v15, v14, v13, v12, v11, v10, v9, v8]

### Modeling a sphere

A sphere is created as a mesh of sixty-four polygons with fifty-eight vertices. It is centered at the origin.



**Figure 4-9** Sphere model.

The X3D <Sphere> node provides information about the radius by its “radius” attribute. A sphere in this project is composed of eight layers of polygons. Each layer has eight polygons. Notice, polygons at the top and bottom layer are triangles but the others are rectangles. We define the top vertex as vertex 0 and the bottom vertex as 57. Polygons in layer 1 are built from vertex 0 and vertices 1-8; polygons in layer 2 are built from vertices 1-8 and vertices 9-16; polygons in layer 3 are built from vertices 9-16 and vertices 17-24; polygons in layer 4 are built from vertices 17-24 and vertices 25-32; polygons in layer 5 are built from vertices 25-32 and vertices 33-40; polygons in layer 6 are built from vertices 33-40 and vertices 41-48; polygons in layer 7 are built from vertices 41-48 and vertices 49-56; polygons in layer 8 are built from vertices 49-56 and vertex 57. The world coordinates of each vertex are illustrated below. Here  $r$  is the radius of the sphere.

Vertex	x coordinate	y coordinate	Z coordinate
v0	0	$r$	0
v1	$r \times \cos(3\pi/8)$	$r \times \sin(3\pi/8)$	0
v2	$r \times \cos(3\pi/8) \times \cos(\pi/4)$	$r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(\pi/4)$
v3	$r \times \cos(3\pi/8) \times \cos(2\pi/4)$	$r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(2\pi/4)$
v4	$r \times \cos(3\pi/8) \times \cos(3\pi/4)$	$r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(3\pi/4)$
v5	$r \times \cos(3\pi/8) \times \cos(4\pi/4)$	$r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(4\pi/4)$
v6	$r \times \cos(3\pi/8) \times \cos(5\pi/4)$	$r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(5\pi/4)$
v7	$r \times \cos(3\pi/8) \times \cos(6\pi/4)$	$r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(6\pi/4)$
v8	$r \times \cos(3\pi/8) \times \cos(7\pi/4)$	$r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(7\pi/4)$
v9	$r \times \cos(2\pi/8)$	$r \times \sin(2\pi/8)$	0
v10	$r \times \cos(2\pi/8) \times \cos(\pi/4)$	$r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(\pi/4)$
v11	$r \times \cos(2\pi/8) \times \cos(2\pi/4)$	$r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(2\pi/4)$
v12	$r \times \cos(2\pi/8) \times \cos(3\pi/4)$	$r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(3\pi/4)$
v13	$r \times \cos(2\pi/8) \times \cos(4\pi/4)$	$r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(4\pi/4)$
v14	$r \times \cos(2\pi/8) \times \cos(5\pi/4)$	$r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(5\pi/4)$
v15	$r \times \cos(2\pi/8) \times \cos(6\pi/4)$	$r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(6\pi/4)$
v16	$r \times \cos(2\pi/8) \times \cos(7\pi/4)$	$r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(7\pi/4)$
v17	$r \times \cos(\pi/8)$	$r \times \sin(\pi/8)$	0
v18	$r \times \cos(\pi/8) \times \cos(\pi/4)$	$r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(\pi/4)$
v19	$r \times \cos(\pi/8) \times \cos(2\pi/4)$	$r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(2\pi/4)$
v20	$r \times \cos(\pi/8) \times \cos(3\pi/4)$	$r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(3\pi/4)$

v21	$r \times \cos(\pi/8) \times \cos(4\pi/4)$	$r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(4\pi/4)$
v22	$r \times \cos(\pi/8) \times \cos(5\pi/4)$	$r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(5\pi/4)$
v23	$r \times \cos(\pi/8) \times \cos(6\pi/4)$	$r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(6\pi/4)$
v24	$r \times \cos(\pi/8) \times \cos(7\pi/4)$	$r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(7\pi/4)$
v25	$r$	$0$	$0$
v26	$r$	$0$	$-r \times \sin(\pi/4)$
v27	$r \times \cos(\pi/4)$	$0$	$-r \times \sin(2\pi/4)$
v28	$r \times \cos(2\pi/4)$	$0$	$-r \times \sin(3\pi/4)$
v29	$r \times \cos(3\pi/4)$	$0$	$-r \times \sin(4\pi/4)$
v30	$r \times \cos(4\pi/4)$	$0$	$-r \times \sin(5\pi/4)$
v31	$r \times \cos(5\pi/4)$	$0$	$-r \times \sin(6\pi/4)$
v32	$r \times \cos(6\pi/4)$	$0$	$-r \times \sin(7\pi/4)$
v33	$r \times \cos(\pi/8)$	$-r \times \sin(\pi/8)$	$0$
v34	$r \times \cos(\pi/8) \times \cos(\pi/4)$	$-r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(\pi/4)$
v35	$r \times \cos(\pi/8) \times \cos(2\pi/4)$	$-r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(2\pi/4)$
v36	$r \times \cos(\pi/8) \times \cos(3\pi/4)$	$-r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(3\pi/4)$
v37	$r \times \cos(\pi/8) \times \cos(4\pi/4)$	$-r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(4\pi/4)$
v38	$r \times \cos(\pi/8) \times \cos(5\pi/4)$	$-r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(5\pi/4)$
v39	$r \times \cos(\pi/8) \times \cos(6\pi/4)$	$-r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(6\pi/4)$
v40	$r \times \cos(\pi/8) \times \cos(7\pi/4)$	$-r \times \sin(\pi/8)$	$-r \times \cos(\pi/8) \times \sin(7\pi/4)$
v41	$r \times \cos(2\pi/8)$	$-r \times \sin(2\pi/8)$	$0$
v42	$r \times \cos(2\pi/8) \times \cos(\pi/4)$	$-r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(\pi/4)$
v43	$r \times \cos(2\pi/8) \times \cos(2\pi/4)$	$-r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(2\pi/4)$
v44	$r \times \cos(2\pi/8) \times \cos(3\pi/4)$	$-r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(3\pi/4)$
v45	$r \times \cos(2\pi/8) \times \cos(4\pi/4)$	$-r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(4\pi/4)$
v46	$r \times \cos(2\pi/8) \times \cos(5\pi/4)$	$-r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(5\pi/4)$
v47	$r \times \cos(2\pi/8) \times \cos(6\pi/4)$	$-r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(6\pi/4)$
v48	$r \times \cos(2\pi/8) \times \cos(7\pi/4)$	$-r \times \sin(2\pi/8)$	$-r \times \cos(2\pi/8) \times \sin(7\pi/4)$
v49	$r \times \cos(3\pi/8)$	$-r \times \sin(3\pi/8)$	$0$
v50	$r \times \cos(3\pi/8) \times \cos(\pi/4)$	$-r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(\pi/4)$
v51	$r \times \cos(3\pi/8) \times \cos(2\pi/4)$	$-r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(2\pi/4)$
v52	$r \times \cos(3\pi/8) \times \cos(3\pi/4)$	$-r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(3\pi/4)$
v53	$r \times \cos(3\pi/8) \times \cos(4\pi/4)$	$-r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(4\pi/4)$
v54	$r \times \cos(3\pi/8) \times \cos(5\pi/4)$	$-r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(5\pi/4)$
v55	$r \times \cos(3\pi/8) \times \cos(6\pi/4)$	$-r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(6\pi/4)$
v56	$r \times \cos(3\pi/8) \times \cos(7\pi/4)$	$-r \times \sin(3\pi/8)$	$-r \times \cos(3\pi/8) \times \sin(7\pi/4)$
v57	$0$	$-r$	$0$

**Table 4-3** World coordinates of a model sphere.

The polygons in the sphere are determined from the vertices as follows.

<b>Polygon</b>	<b>A set of vertices arranged in counter-clockwise order</b>
polygon0	[v0, v1, v2]
polygon1	[v0, v2, v3]
polygon 2	[v0, v3, v4]
polygon 3	[v0, v4, v5]
polygon 4	[v0, v5, v6]

polygon 5	[v0, v6, v7]
polygon 6	[v0, v7, v8]
polygon 7	[v0, v8,v1]
polygon 8	[v1, v9, v10, v2]
polygon 9	[v2, v10, v11, v3]
polygon 10	[v3, v11, v12, v4]
polygon 11	[v4, v12, v13, v5]
polygon 12	[v5, v13, v14, v6]
polygon 13	[v6, v14, v15, v7]
polygon 14	[v7, v15, v16, v8]
polygon 15	[v8, v16, v9, v1]
polygon 16	[v9, v17, v18, v10]
polygon 17	[v10, v18, v19, v11]
polygon 18	[v11, v19, v20, v12]
polygon 19	[v12, v20, v21, v13]
polygon 20	[v13, v21, v22, v14]
polygon 21	[v14, v22, v23, v15]
polygon 22	[v15, v23, v24, v16]
polygon 23	[v16, v24, v17, v9]
polygon 24	[v17, v25, v26, v18]
polygon 25	[v18, v26, v27, v19]
polygon 26	[v19, v27, v28, v20]
polygon 27	[v20, v28, v29, v21]
polygon 28	[v21, v29, v30, v22]
polygon 29	[v22, v30, v31, v23]
polygon 30	[v23, v31, v32, v24]
polygon 31	[v24, v32, v25, v17]
polygon 32	[v25, v33, v34, v26]
polygon 33	[v26, v34, v35, v27]
polygon 34	[v27, v35, v36, v28]
polygon 35	[v28, v36, v37, v29]
polygon 36	[v29, v37, v38, v30]
polygon 37	[v30, v38, v39, v31]
polygon 38	[v31, v39, v40, v32]
polygon 39	[v32, v40, v33, v25]
polygon 40	[v33, v41, v42, v34]
polygon 41	[v34, v42, v43, v35]
polygon 42	[v35, v43, v44, v36]
polygon 43	[v36, v44, v45, v37]
polygon 44	[v37, v45, v46, v38]
polygon 45	[v38, v46, v47, v39]
polygon 46	[v39, v47, v48, v40]
polygon 47	[v40, v48, v41, v33]
polygon 48	[v41, v49, v50, v42]
polygon49	[v42, v50, v51, v43]
polygon50	[v43, v51, v52, v44]
polygon51	[v44, v52, v53, v45]
polygon52	[v45, v53, v54, v46]
polygon53	[v46, v54, v55, v47]
polygon 54	[v47, v55, v56, v48]

polygon 55	[v48, v56, v49, v41]
polygon5 6	[v49, v57, v50]
polygon 57	[v50, v57, v51]
polygon58	[v51, v57, v52]
polygon 59	[v52, v57, v53]
polygon 60	[v53, v57, v54]
polygon 61	[v54, v57, v55]
polygon 62	[v55, v57, v56]
polygon 63	[v56, v57, v49]

**Table 4-4** Polygon surfaces with their assigned vertices of a model sphere.

Our translator consists of five core classes: super-class Primitive, sub-class Box, Cone, Cylinder, and Sphere. Each has a method `createFirstWC()` used to generate the world coordinates of that object. Below is sample JavaScript code for the `Box.createFirstWC()` method.

```
Box.prototype.createFirstWC = function( )
{
    var wc = this.wc;      // an array of world coordinates
    // x, y, z is the dimension of a box.
    var x = this.x; var y = this.y; var z = this.z;
    /* Model world coordinates of a box. */
    wc[0].setXYZ(-x/2, y/2, z/2);   wc[1].setXYZ(x/2, y/2, z/2);
    wc[2].setXYZ(x/2, -y/2, z/2);   wc[3].setXYZ(-x/2, -y/2, z/2);
    wc[4].setXYZ(-x/2, y/2, -z/2);  wc[5].setXYZ(x/2, y/2, -z/2);
    wc[6].setXYZ(x/2, -y/2, -z/2);  wc[7].setXYZ(-x/2, -y/2, -z/2);
}
```

Code of this method for Cone, Cylinder, and Sphere is in *Appendix B*

## 4.8 Viewing Transformation

After we get the world-coordinate description, the next step in the viewing pipeline is to perform viewing-transformation operations to convert world coordinates to the viewing coordinates. Generally, users can change perspectives by turning around the x, y, or z axis, moving left or right to the objects, moving up or down to the objects, moving close

to or away from the objects, or making a combination of these. Viewing coordinates will be converted from the world-coordinate positions according to the perspective users have chosen. World coordinates and viewing coordinates can be represented with 4 by 4 matrices. There is a composition transformation matrix for transferring the world-coordinate system into the viewing-coordinate system as shown in the equation below.

$$VC = M_{wc,vc} \bullet WC$$

VC is a matrix representing viewing coordinates and WC is a matrix representing world coordinates.  $M_{wc,vc}$  is a composition transformation matrix.  $M_{wc,vc}$  is superimposed from the general sequence of rotate-translate transformations.

$$M_{wc,vc} = R_z \bullet R_x \bullet R_y + T$$

$R_x$ ,  $R_y$ , and  $R_z$  are rotation transformation matrices for rotating objects about the x-, y-, and z-axis. T is a translation matrix. Let x, y, and z represent a rotation angle when rotating object about the x, y and z-axis respectively. Let  $t_x$ ,  $t_y$ , and  $t_z$  represent a translation distance in the x, y, and z direction respectively.

$$R_z = \begin{bmatrix} \cos z & -\sin z & 0 & 0 \\ \sin z & \cos z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos x & -\sin x & 0 \\ 0 & \sin x & \cos x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos y & 0 & \sin y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin y & 0 & \cos y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z \bullet R_x \bullet R_y = \begin{bmatrix} \cos y \cos z - \sin x \sin y \sin z & -\cos x \sin z & \sin y \cos z - \sin x \cos y \sin z & 0 \\ \cos y \sin z + \sin x \sin y \cos z & \cos x \cos z & -\sin y \sin z + \sin x \cos y \cos z & 0 \\ \cos x \sin y & -\sin x & \cos x \cos y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{wc,vc} = R_z \bullet R_x \bullet R_y + T = \begin{bmatrix} \cos y \cos z - \sin x \sin y \sin z & -\cos x \sin z & \sin y \cos z - \sin x \cos y \sin z & t_x \\ \cos y \sin z + \sin x \sin y \cos z & \cos x \cos z & -\sin y \sin z + \sin x \cos y \cos z & t_y \\ \cos x \sin y & -\sin x & \cos x \cos y & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformation matrix is a 4 by 4 matrix, not a 3 by 3 matrix. This is because homogeneous coordinates are used. Since many graphics applications involve sequences of geometric transformations, such transformation sequences can be efficiently processed if we reformulate a matrix representation by expanding a 3 by 3 matrix to a 4 by 4 matrix. In this project, for each vertex, we calculate viewing-coordinate positions by multiplying the  $M_{wc,vc}$  matrix with the WC matrix.

The output document has twelve buttons for users to view a scene from different perspectives. Whenever one of the buttons is clicked, we need to recalculate the viewing transformation to get new viewing coordinates of the scene in that perspective. More specifically, when the *yaw left*, *yaw right*, *pitch in*, *pitch out*, *roll left*, or *roll right* buttons are clicked, the rotation angle of the x, y, z-axis will be changed by ten degrees in the appropriate direction. When the *move left*, *move right*, *move up*, *move down*, *zoom in*, or *zoom out* buttons are clicked, a translation by ten units is applied to the x, y, z-axis.

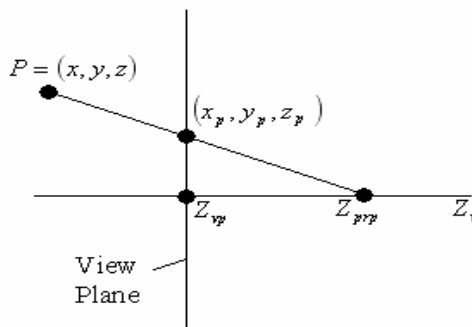


The class method to perform this task is `Primitive.createVC()`. The `Primitive.createVC()` method converts a world-coordinate description to viewing coordinates. Below is JavaScript code of this method.

```
Primitive.prototype.createVC = function()
{
    // Get transformation matrix  $M_{wc,vc}$ 
    var matrix = perspective.matrix;
    var wc = this.wc;           // an array of world coordinates
    var vc = this.vc;           // an array of viewing coordinates
    for(var i=0; i<this.num_vertices; i++)
    {
        /* Calculate viewing coordinates,  $VC = M_{wc,vc} \bullet WC$  */
        vc[i].setXYZ(wc[i].x3D, wc[i].y3D, wc[i].z3D);
        vc[i].rotate_translate(matrix);
    }
}
```

## 4.9 Perspective Projection

Once world-coordinate descriptions of the objects in a scene are converted to viewing coordinates, we can project the 3D objects onto the 2D view plane. We apply a perspective projection to transform viewing-coordinate descriptions to projected coordinates.



The position  $P$  in the picture can be considered as a viewing coordinate.  $(x_p, y_p, z_p)$  is a projected coordinate.

$$x_p = x \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) = x \left( \frac{d_p}{z_{prp} - z} \right)$$

$$y_p = y \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) = y \left( \frac{d_p}{z_{prp} - z} \right)$$

In this project, we use the xy plane as the view plane, so the  $z_p$  is equal to zero. We use (0, 0, 300) as the projection reference point. We use the SVG <polygon> element to display polygons. This element lets us specify a series of points that describe a polygon area. The “points” attribute of the <polygon> tag consists of a series of x- and y-coordinate pairs. In our case, a series of xy projection coordinate pairs that describes a polygon area is ascribed to the “points “ attribute for creating each polygon.

The calculations of projections are done by the `Primitive.projection()` method. This method is to compute projected positions which will be supplied to <polygon> tags later to display polygon areas on the scene.

```
Primitive.prototype.createProjection = function()
{
    var vc = this.vc; // an array of viewing coordinates
    var projection = this.projection; // an array of projection
    /* Calculate projected positions for every polygon vertex. */
    for(var i=0; i<this.num_vertices; i++)
    {
        projection[i].setXYZ(vc[i].x3D, vc[i].y3D, vc[i].z3D);
        /*
        * projection( ) is to compute a projection of each polygon
        * vertex as described above.
        */
        projection[i].project(viewPoint);
    }
}
```

## 4.10 3D Geometric Transformations

The <Transform> node that we support has three basic geometric transformations, translation, rotation, and scale. After we apply the transformation to objects, it will alter the world-coordinate descriptions of the objects. So, if in the X3D source document, a primitive node - <Box>, <Cone>, <Cylinder>, <Sphere>- has a parent <Transform> node, the model world coordinates of the object will be altered to new positions. There are matrix formulations for translation, rotation, and scale to efficiently alter the world-coordinate descriptions as illustrated in the equations below.

$$WC = T \bullet WC \quad \text{eq1}$$

$$WC = R_{xyz} \bullet WC \quad \text{eq2}$$

$$WC = S \bullet WC \quad \text{eq3}$$

eq1, eq2, and eq3 are the transformation equations for a translation, rotation, and scale transformation respectively. Newly transformed world coordinates,  $WC$ , can be computed by the multiplication of the transformation matrix and an old world-coordinate description,  $WC$ .

$T$  is a translation formulation matrix.

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The parameters  $t_x$ ,  $t_y$ , and  $t_z$  are translation distances in the x, y, and z direction. We get these values from the “translation” attributes of a <Transform> node.

$R_{xyz}$  is a formulation matrix of a composite rotation.

$$R_{xyz} = R_z \bullet R_x \bullet R_y = \begin{bmatrix} \cos y \cos z - \sin x \sin y \sin z & -\cos x \sin z & \sin y \cos z - \sin x \cos y \sin z & 0 \\ \cos y \sin z + \sin x \sin y \cos z & \cos x \cos z & -\sin y \sin z + \sin x \cos y \cos z & 0 \\ \cos x \sin y & -\sin x & \cos x \cos y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

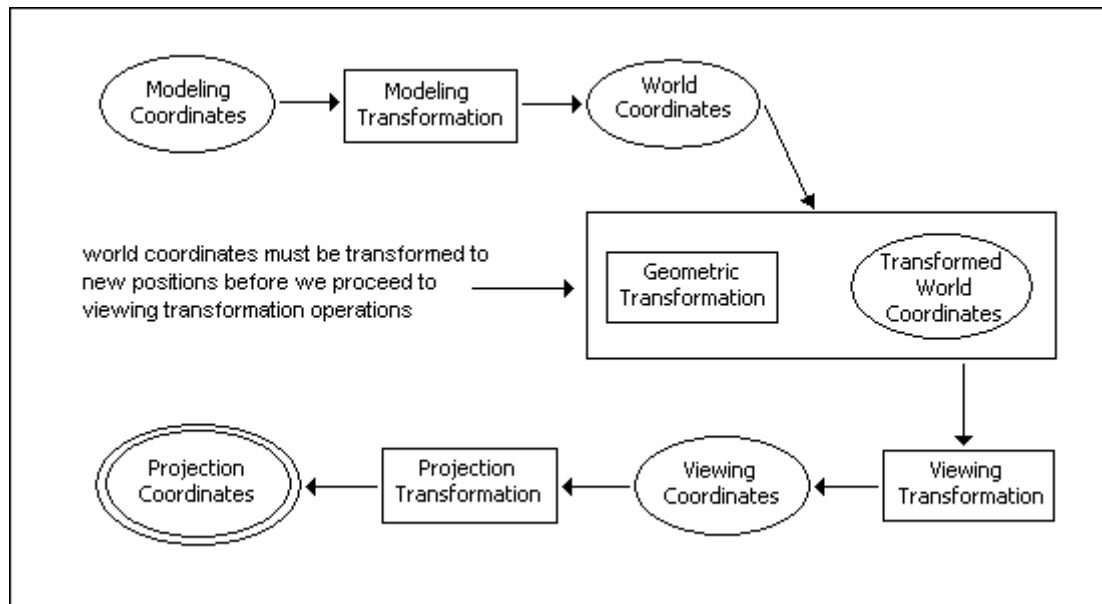
The parameters x, y, and z are rotation angles when objects are rotated about the x, y, and z axis respectively. They are obtained from the “rotation” attribute of a <Transform> node.

S is a scale formulation matrix.

$$s = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

the parameters  $s_x$ ,  $s_y$ , and  $s_z$  are scaling factors of the x, y, and z axis. We obtain the scaling factors from the “scale” attribute of a <Transform> node.

If the source X3D document specifies that an object needs to be transformed, the model world-coordinate description of the object must be transformed to a new description before we perform view transformation to convert world-coordinate positions to viewing-coordinate positions.



**Figure 4-10** Diagram of perform geometric transformation.

The geometric transformation process returns newly transformed world coordinates. This needs to be done before we perform a viewing transformation.

The geometric transformations are performed by `Primitive.modifyWC()`. This method calculates new world coordinates of an object if the object needs to be geometrically transformed. This method gets its transformation information by backtracking from the `<Shape>` node until it hits the `<Scene>` node. Below is a pseudo code for this method.

```

Primitive.prototype.modifyWC = function()
{
    var wc = this.wc;
    /* Keep tracking up until hit the "Scene*/
    while(parentNode.nodeName != "Scene")
    {
        if(parentNode.nodeName == "Transform")
        {

```

```

    /* Get translation, rotation, and scale values. */
    var translation = parentNode.getAttribute("translation");
    var rotation = parentNode.getAttribute("rotation");
    var scale = parentNode.getAttribute("scale");

    /*
    Get translation distance Tx, Ty, Tz from the
    translation attribute.
    */
    /*
    Get rotation angle Rx, Ry, Rz from the rotation attribute.
    */
    /* Get scale factor Sx, Sy, Sz from the scale attribute. */

    /* Get a T*R transformation matrix. */
    var transform = new Transform();
    var matrix = transform.matrix;
    transform.translate(Tx, Ty, Tz);
    transform.rotate(Rx, Ry, Rz);

    /* Calculate new WC = R*T* S*WC. */
    for(var i=0; i<this.num_vertices; i++)
    {
        /* Get S*WC. */
        wc[i].scale(Sx, Sy, Sz)
        wc[i].translate(matrix);
        wc[i].rotate(matrix);
    }
}
}
}

```

## 4.11 Illumination Model

Realistic displays of a scene are obtained by generating perspective projections of objects, which are obtained from the viewing pipeline process, and by applying natural lighting effects to visible surfaces. An illumination model, also called a lighting model, is used to calculate the intensity of light we should see at a given point on the surface of an object. Illumination models in computer graphics are often loosely derived from the physical laws that describe surface light intensities. The Phong lighting model is the simplest and by far the most popular illumination model for 3D computer graphics. In

this project, we use this model to calculate the light intensity at a given point on the polygon surface. The Phong lighting model considers only the effects of a single light source, not multiple sources shining directly onto a surface that are then reflected directly on to the viewpoint. Below is the equation to calculate the intensity of a given point in the Phong lighting model.

$$I = I_a + I_d + I_s + I_e$$

The first term  $I_a$  is ambient light. Ambient light is light that arrives equally from all directions, rather than from the direction of one light source. The formula for the intensity of the outgoing ambient light is

$$I_a = \rho_a I_a^{in}$$

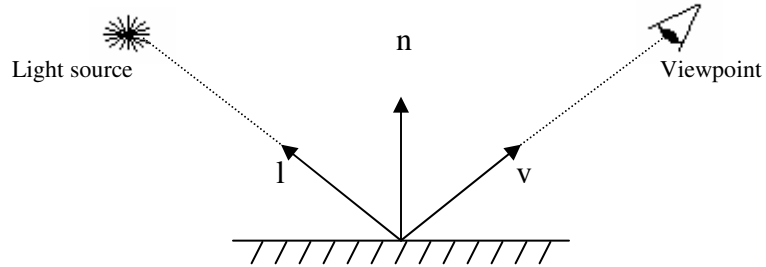
$\rho_a$  is a constant value called an ambient reflectivity coefficient that controls the amount of ambient light reflected from the surface.  $I_a^{in}$  represents the total intensity of the incoming ambient light.

The second term  $I_d$  is diffused light. Diffusely reflected light is light which is reflected evenly in all directions away from the surface. The amount of light, which is diffusely reflected, is modeled as

$$I_d = \rho_d I_d^{in} (l \bullet n)$$

$I_d^{in}$  is the intensity of the incoming diffused light. The value  $\rho_d$  is a constant, which is called the diffused reflectivity coefficient of the surface. This value represents a physical property of the surface material.  $l$  is a unit vector representing the direction of the point

light source.  $n$  is a unit surface of a normal vector. Unit vector  $l$  and  $n$  are displayed in the picture below.



**Figure 4-11** The fundamental vectors of the Phong lighting model.

The third term  $I_s$  is specular light. Specular light is light from a point light source, which will be reflected specularly. The purpose of specular reflection is to model shiny surfaces. The Phong formula for the intensity  $I_s$  of specularly reflected light is

$$I_s = \rho_s I_s^{in} (h \cdot n)^f$$

$$h = \frac{l + v}{\|l + v\|}$$

$\rho_s$  is a constant called the specular reflectivity coefficient to control the amount of specular reflection.  $f$  is a specular exponent to control the shininess of the surface by controlling the narrowness of the spread of specularly reflected light. The exponent  $f$  must be more than or equal to 1. Higher exponent values make the specular highlights smaller and the surface appear shinier.  $l$  is a unit vector representing the direction of the point light source and  $v$  is a unit surface's normal vector.

The last term  $I_e$  is the emissive intensity constant. This is equal to the intensity of the light emitted by the surface in addition to the reflected light.



Putting the different lights together, we get a total outgoing light intensity that equals to

$$I = \rho_a I_a^{in} + \rho_d I_d^{in} (l \bullet n) + \rho_s I_s^{in} (h \bullet n)^f + I_e$$

We can assign arbitrary numbers to a constant  $\rho_a$ ,  $\rho_d$ ,  $\rho_s$ , and  $f$  under one condition, that is the summation of three coefficients  $\rho_a$ ,  $\rho_d$ , and  $\rho_s$  must be equal to 1. After experimenting using many values to determine which values produce smooth shades onto polygon surfaces, we found that assigning 0.2 to  $\rho_a$ , 0.6 to  $\rho_d$ , 0.2 to  $\rho_s$ , and 2 to  $f$  can produce decent shading. We used these constant values to calculate light intensity in this project. To minimize intensity calculations in the rendering process, we assigned 1 to  $I_a^{in}$ ,  $I_d^{in}$ ,  $I_s^{in}$ , and 0 to  $I_e$ . In this application, we also set up the light source as coming from the same direction as the viewpoint. Hence, the unit vector  $l$  is equal to the unit vector  $v$ . Notice the intensity value calculated in this equation varies from 0 to 1. The intensity equation used in our program after replacing with assigned constant values is

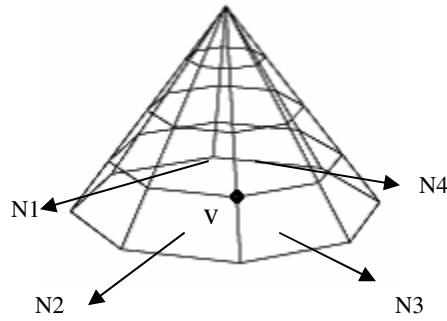
$$I = 0.2 + 0.6(l \bullet n) + 0.2(h \bullet n)^2$$

#### 4.12 Gouraud Shading Model

The intensity we get from the illumination model is used to determine the light intensity for all projected positions for the various surfaces in the scene. Surface rendering can be accomplished by interpolating intensities from these illumination-model calculations across surfaces. To avoid confusion, the model for calculating light intensity at a single surface point is referred to as an illumination model or lighting model. Surface rendering or surface shading is a procedure for applying a light model to obtain intensities for all

the projected surface positions in a scene. In this project, we use the “Gouraud” surface-shading model. “Gouraud” shading is an intensity-interpolation scheme. This scheme renders a polygon surface by linearly interpolating intensity values across a surface. Each polygon surface is rendered with “Gouraud” shading by performing the following calculation

1. Determine the average unit normal vector at each polygon vertex. At each vertex of a polygon, we obtain a normal vector by averaging the surface normals of all polygons sharing that vertex as illustrated in the picture below.



For any vertex position V, we obtain the unit vertex normal with the calculation

$$N_v = \frac{\sum_{k=1}^n N_k}{\left| \sum_{k=1}^n N_k \right|}$$

2. Apply the illumination model to each vertex to calculate that vertex's intensity.

Once we have the normal vector, we can determine the intensity at the vertices from the lighting model. Referring back to the intensity equation in the

illumination model,  $I = \rho_a I_a^{in} + \rho_d I_d^{in} (l \bullet n) + \rho_s I_s^{in} (h \bullet n)^f + I_e$  or

$I = 0.2 + 0.6(l \bullet n) + 0.2(h \bullet n)^2$  (We ran several coefficient numbers and found that

assigning 0.2 to  $\rho_a$ , 0.6 to  $\rho_d$ , 0.2 to  $\rho_s$ , and 2 to  $f$  produce smooth shading on a polygon surface). Here  $n$  is the average unit normal vector at each polygon vertex.

3. Linearly interpolate the vertex intensities over the surface of the polygon. Fortunately, SVG has `<linearGradient>` tags to do this job for us. However, this tag requires us to specify a starting and stopping color along the polygon edge. So it is our job to calculate starting and stopping color intensity, and then supply it to the `<linearGradient>` tag. In the next section we will describe how to calculate the color intensity along the polygon edge.

The process of Gouraud shading is done by the `Primitive.findIntensity()` method. It calculates the light intensity of every polygon vertex. Below is pseudo code of this method.

```
Primitive.prototype.findIntensity = function()
{
    /*
    * Calculate a unit vector l from the center of the object and a
    * viewpoint position.
    */
    var centerPoint = this.center2;
    var l = new Vector(centerPoint.x3D, centerPoint.y3D,
                      centerPoint.z3D - viewPoint);
    l.normalize();

    /* Calculate a unit vector h =  $\frac{l+v}{|l+v|}$  */
    var h = new Vector(l.A, l.B, l.C);
    h.addVector(l);
    h.normalize();

    var n = this.avgNormalVectors; // an array of normal vectors
    var I = this.intensity;
    var pa = 0.2;
    var pd = 0.6;
    var ps = 0.2;
```

```

var p = this.projection;

/* Calculate the light intensity for every polygon vertex. */
for(var i=0; i<this.num_vertices; i++)
{
    var Id = new Vector(l.A, l.B, l.C);
    l_dot_n = Id.dotVector(n[i]);
    var Is = new Vector(h.A, h.B, h.C);
    h_dot_n = Is.dotVector(n[i]);
    I[i] = pa + pd*l_dot_n + ps*(h_dot_n * h_dot_n);
}
}

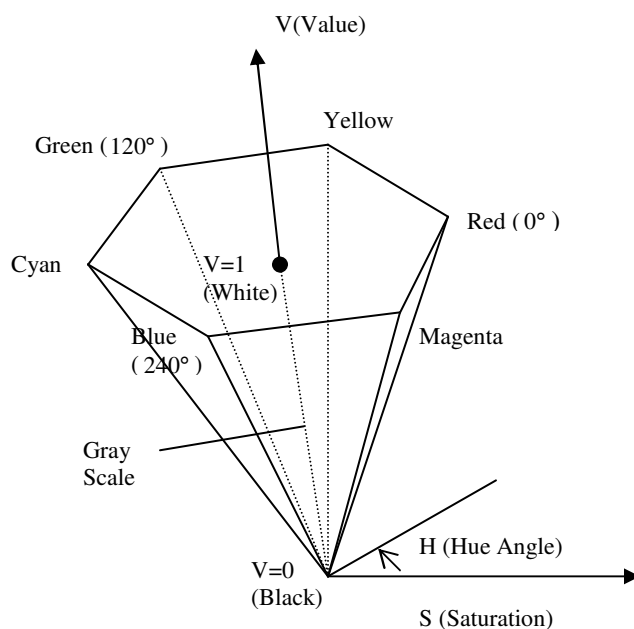
```

### 4.13 HSV Color Model

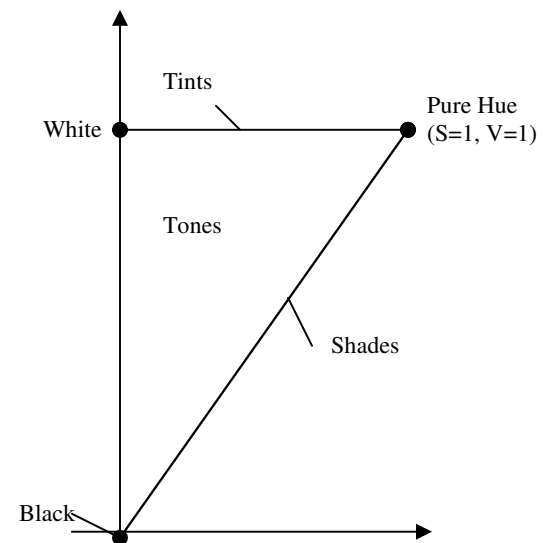
After we model the light intensity of all projected positions onto a polygon surface, the next step is to find a color or shade we should paint on those surfaces. Every object in the scene already has an assigned color, which is specified by the user or assigned to the default color by X3DToSVG.dtd. The assigned color is in RGB (Red Green Blue) format. We use the light intensity of a polygon vertex together with the assigned color of the object to determine a tone for that vertex. However, it is difficult to calculate a new tone of a given vertex using an RGB color. To do this, another color model, called the HSV (Hue Saturation Value) model, is used. This model has a more intuitive color concept. We compute the color of all surface positions on the scene by using HSV rather than RGB, and it is not too difficult to convert colors back and forth between these two formats.

The HSV model uses color descriptions that have a more intuitive appeal to us. To give a color specification, we select a spectral color and the amounts of white and black that are added to form various shades, tints, and tones in the scene. Starting with the pigment for

a “pure color” (or “pure hue”), which – in this case- is an object’s color, we add a black pigment to produce different shades of that color. The more black pigment, the darker the shade. Similarly, different tints of the color are obtained by adding a white pigment to the original color, making it lighter as more white is added. Tones of the color are produced by adding both black and white pigments. Color parameters in this model are hue (H), saturation (S), and value (V). The relation of parameters H, S, and V can be represented in the HSV hexcone and in a cross section of the HSV hexcone as displayed below.



**Figure 4-12** The HSV hexcone.



**Figure 4-13** Cross section of the HSV hexcone, showing regions for shades, tints, and tones.

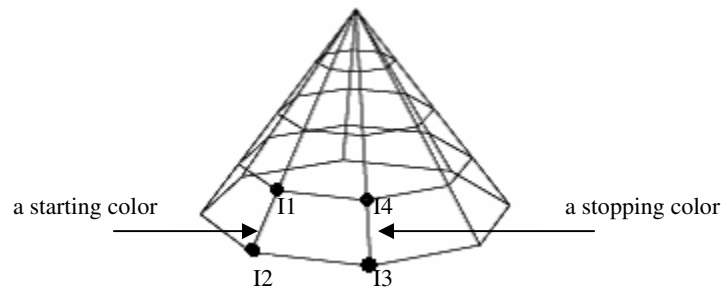
Hue is represented as an angle around the vertical axis, ranging from  $0^\circ$  at red through  $360^\circ$ . Complementary colors are  $180^\circ$  apart; for example, cyan is a complementary color

of red. Value S and V varies from 0 to 1. This is more intuitive for most users. Starting with a selection for a pure hue, which specifies the hue angle H and set  $V=S=1$ , we describe the color we want in terms of adding either white or black to the pure hue. Adding black decreases the setting for V while S is held constant. Similarly, when white is to be added to the hue selected, parameter S is decreased while keeping V constant. By adding some black and some white, we decrease both V and S.

We apply this model to calculate a color along a polygon edge known as a starting and stopping color, which will be supplied to an SVG `<linearGradient>` tag to performs linear interpolation over a polygon surface. The procedure to compute a color along a polygon edge is

1. Get the object's color from a source document, which is in RGB format. Convert it into HSV format. (We have covered JavaScript code for converting RGB to HSV and vice versa in *Appendix B*)
2. Determine a color of a given point by adjusting only the S value and keeping the H and V values constant. The light intensity we get from the illumination model can be thought of as equivalent to S. The lower light intensity, the more white is added to a pure color or hue of the object. In our implementation, we directly replace the S value with half of the light-intensity value to get a new HSV tone at a given position (after testing many proportions for the light intensity, we found that replacing S with half of the light intensity value generates decent tones for a color). However, what we

actually want in this case is a starting and stopping color along the polygon edge. So the intensity used to determine a starting and stopping tone is the average intensity of the two end points of that edge.



**Figure 4-14** The average intensity of polygon edges.

In the above picture, the intensity on the left edge is the average of I1 and I2 and the intensity on the right edge is the average of I3 and I4.

3. Then we convert the HSV color back to an RGB color because the SVG `<linearGradient>` tag accepts only RGB-format colors.

The starting and stopping color are computed by the `findColor()` method. Every subclass Box, Cone, Cylinder, and Sphere has this method. Below is JavaScript code for `Box.findColor()`. You can find the whole code of this method for the other subclasses in *Appendix B*.

```
Box.prototype.findColor = function()
{
    /*
    * The light intensity of every polygon vertex computed from
    * Primitive.findColor( ) method
    */
    var I = this.intensity;
    /* Calculate the average intensity of each edge. */
}
```

```

var avgI = this.avgI;
avgI[0] = (I[0] + I[3])/2;
avgI[1] = (I[1] + I[2])/2;
avgI[2] = (I[5] + I[6])/2;
avgI[3] = (I[4] + I[7])/2;
avgI[4] = (I[4] + I[0])/2;
avgI[5] = (I[5] + I[1])/2;
avgI[6] = (I[3] + I[7])/2;
avgI[7] = (I[2] + I[6])/2;

/* Get the rgb color and translate into hsv format. */
var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

var front = 0; var back = 1; var left = 2;
var right = 3; var top = 4; var bottom = 5;

var color_left = this.color_left; // starting colors
var color_right = this.color_right; // stopping colors

/* Colors on each edge are in hsv format. */
color_left[front] = hsv[2] - avgI[0]*(0.5);
color_right[front] = hsv[2] - avgI[1]*(0.5);
color_left[back] = hsv[2] - avgI[2]*(0.5);
color_right[back] = hsv[2] - avgI[3]*(0.5);
color_left[left] = hsv[2] - avgI[3]*(0.5);
color_right[left] = hsv[2] - avgI[0]*(0.5);
color_left[right] = hsv[2] - avgI[1]*(0.5);
color_right[right] = hsv[2] - avgI[2]*(0.5);
color_left[top] = hsv[2] - avgI[4]*(0.5);
color_right[top] = hsv[2] - avgI[5]*(0.5);
color_left[bottom] = hsv[2] - avgI[6]*(0.5);
color_right[bottom] = hsv[2] - avgI[7]*(0.5);
}

```

#### 4.14 Visible-Surface Detection

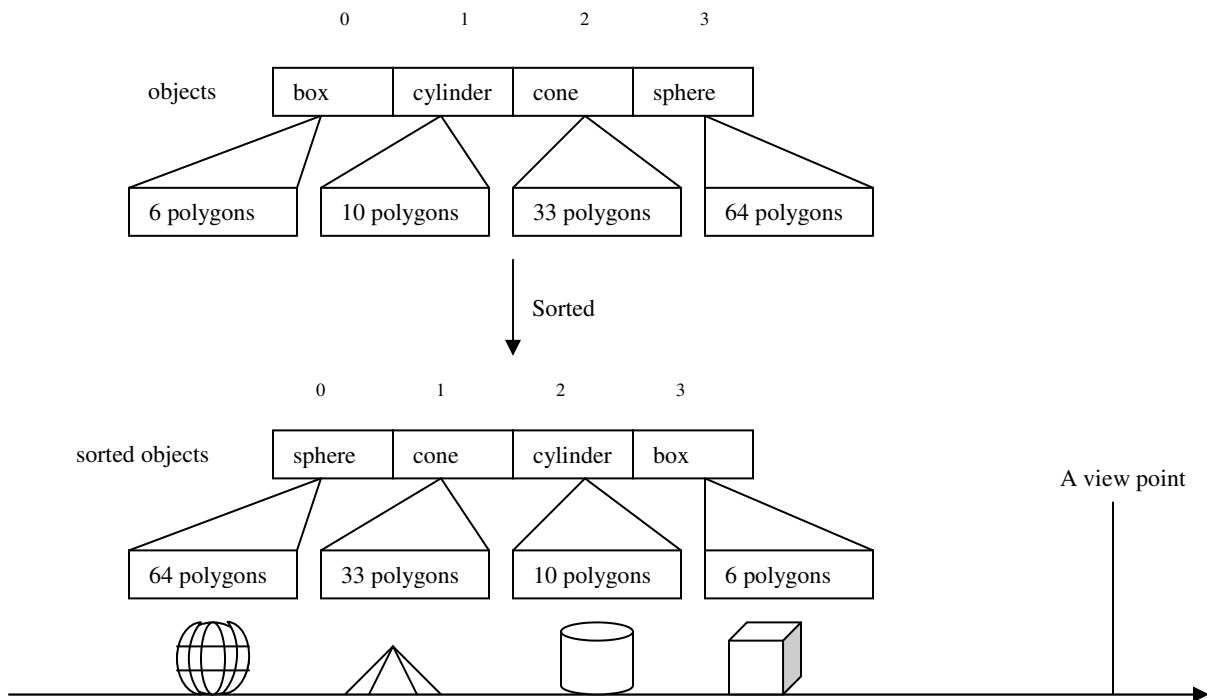
A major consideration in the generation of realistic graphics displays is to identify those parts of a scene that are visible from a chosen viewing position. The various algorithms to solve this problem are referred to as visible-surface detection methods. Visible-surface detection algorithms are broadly classified according to whether they deal with object definitions directly or with their projected images. These two approaches are called object-space methods and image-space methods respectively. An object-space method compares objects and parts of objects to each other within the scene definition to



determine which surfaces, as a whole, we should label as visible. In an image-space algorithm, visibility is decided point by point at each pixel position on the projection plane. Most visible-surface algorithms use image-space methods; however, only object-space methods are applicable to this project because we cannot use SVG to internally describe a polygon pixel by pixel; a polygon is rendered by the SVG engine as a whole. The technique we use in this project for detecting visible surfaces is similar to a painter's algorithm or a depth-sorting algorithm, which is using both image-space and object-space operations.

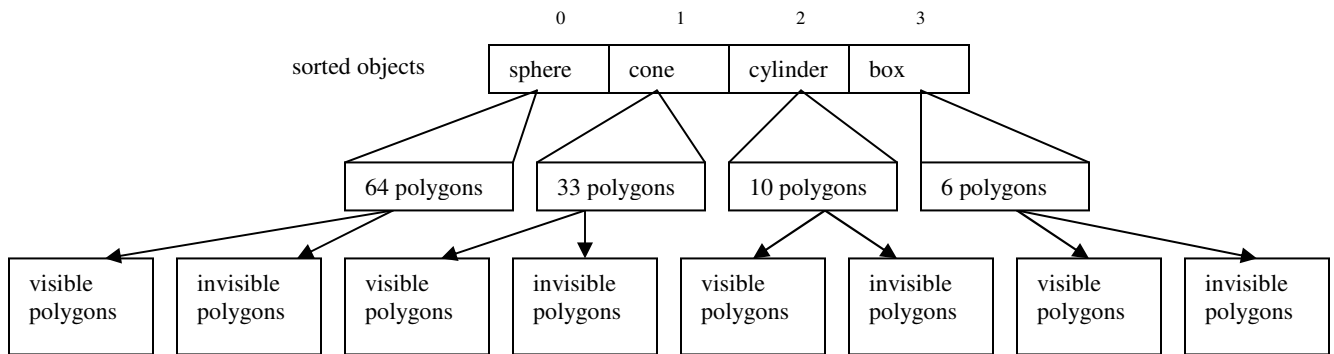
The technique, explained step by step, used in this project:

Step 1: Instantiate all objects of the scene in our program and compare each object with the other objects in the scene to determine which object is to be painted first, second, and so on. The rendering order is arranged according to the object's depth. We use an object's center to determine the depth of each object. The object instantiated in our program can be considered a collection of polygons. Suppose a box is the nearest object to the viewpoint, a sphere is the farthest one from a viewing point, and in between are a cylinder and a cone. The correct rendering order is sphere, cone, cylinder, and box. Of course, the sphere must be drawn first because it is the deepest, when compared to the other objects in the scene and the box should be rendered last because it is closer to the viewpoint than any other objects in the scene.



**Figure 4-15** Arranging objects according to their depth.

Step 2: Determine whether a polygon is visible or invisible and categorize it into one of two groups, either the visible or the invisible group. A back-face detection algorithm is used to identify a back face -which appears invisible from a chosen viewing position- and a front face - which appears visible from a viewpoint. This back-face detection algorithm is used to identify which polygons are visible.



**Figure 4-16** Separate polygons into two groups.

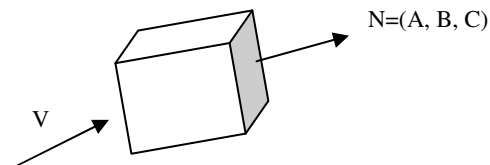
### The back-face detection algorithm

This algorithm is a fast and simple object-space method for identifying the back faces of a polyhedron. We simplify this test further by considering the normal vector  $N$  to a polygon surface, which has Cartesian components  $(A, B, C)$ . To find the normal vector  $N$  to a polygon surface see *Appendix A*. In general, if  $V$  is a vector in the viewing direction from the viewing position, then this polygon is a back face if the dot product of  $V$  and  $N$  is greater than zero,

$$V \cdot N > 0$$

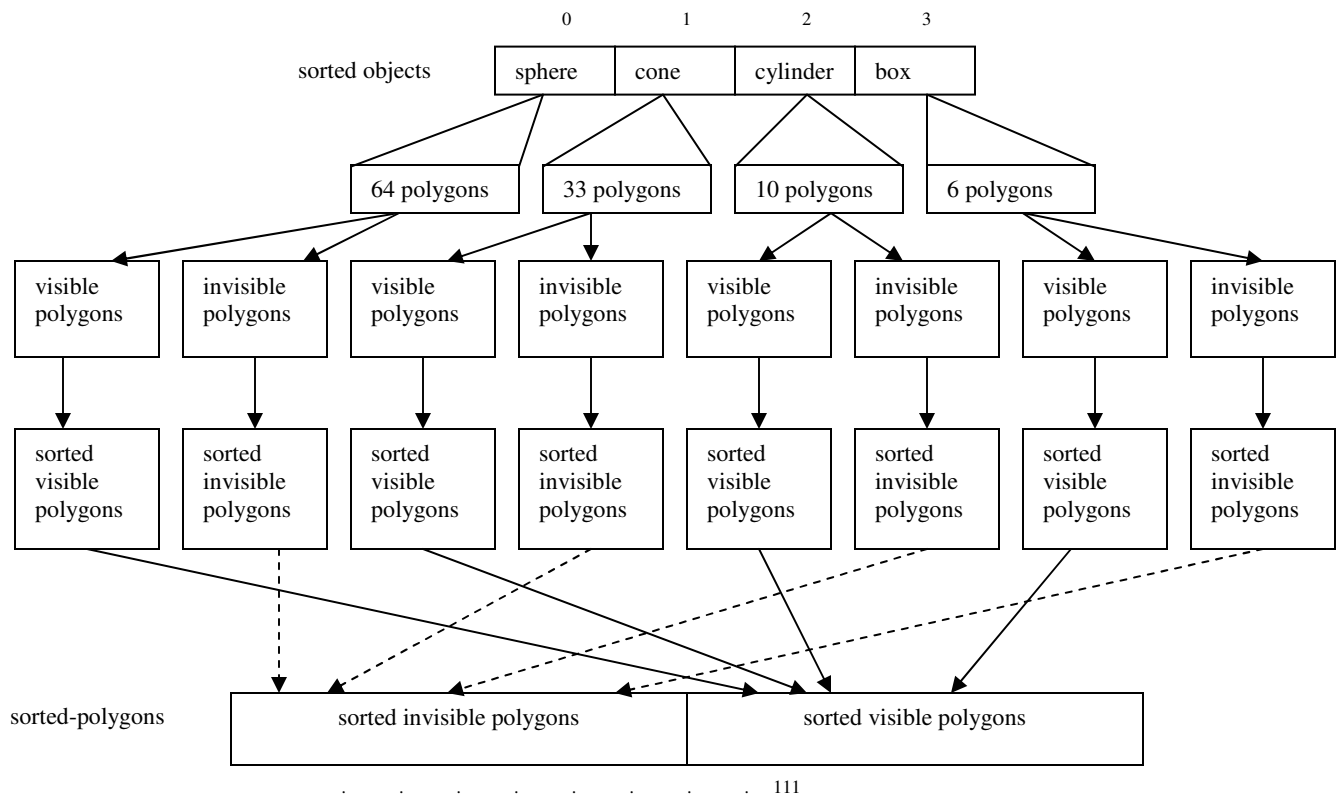
$$\text{or } V \cdot N = |V| |N| \cos \theta > 0$$

Here  $\theta$  is an angle between  $V$  and  $N$ . If  $0^\circ < \theta < 90^\circ$ , or  $180^\circ < \theta < 360^\circ$ , the polygon is a back face and appears invisible from the viewing direction.



Vector  $V$  in the viewing direction and a back-face normal vector  $N$  of a polygon

Step 3: We sort the polygons of the same group according to their depths. Then we gather polygons from different visible groups and do the same thing for invisible groups as well. Therefore, all visible groups are combined together to form a new merged visible group as are the invisible groups. We use the quicksort algorithm to arrange the rendering order. Once we get both newly merged visible and invisible groups sorted, the group of invisible polygons is arranged to be drawn first, and the group of visible polygons is drawn second (The optimized version, which explained in Section 5, will render only visible polygons).



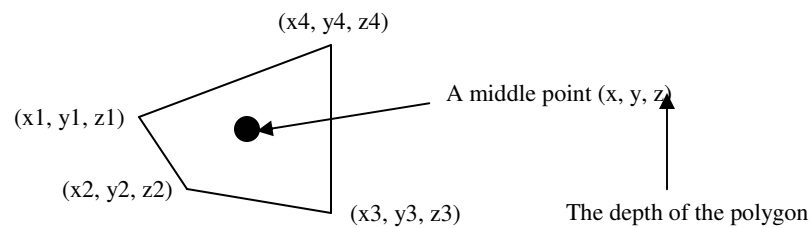
**Figure 4-17** Steps for sorting an rendering order.

Sorted-polygons [0] is the deepest invisible polygon in the scene and will be drawn first.

Sorted-polygons [111] is the closet visible polygon in the scene and will be drawn last.

### A polygon's depth

We estimate a polygon's depth by the z-coordinate of the middle point on the polygon's surface.



**Figure 4-18** The z value of a center point represented as the depth of this polygon.

The z-value of the middle point is computed by averaging z-values of all the polygon's vertices i.e.,  $(z_1+z_2+z_3+z_4)/4$ .

The code that we use to determine the rendering order is done in `drawObjects( )`.

Below is the pseudo code of this function.

```
/*
 * This function is to render every object in the scene and called
 * whenever the perspective is changed.
 */
function drawObjects( )
{
    /*
     * Create a transform matrix of a current perspective. Before
     * this function is called, a globalvariable g_perspective is
     * initiated as Transform object. g_tx, g_ty, g_tz, g_rx, g_ry,
     * and g_rz are positions and angles of the current perspective.
     */
}
```

```

g_perspective.translate(g_tx, g_ty, g_tz);
g_perspective.rotate((g_rx % 360)*0.017453293,
    (g_ry % 360)*0.017453293, (g_rz % 360)*0.01745329);

/*
 * Get viewing coordinates and projections of the objects in the
 * scene.
 */
var g_obj_len = g_objects.length;
for(var i=0; i<g_obj_len; i++)
    g_objects[i].getVCandProjection() ;

/* Calculate color intensity.*/
for(var i=0; i<g_obj_len; i++)
    g_objects[i].getShade( );

/*
 * Arrange the order for drawing objects according to the depth
 * of their center.
 */
quicksortCenter(g_objects, 0, g_obj_len - 1);

/* Create polygons to form the primitives in the scene. */
for(var i=0; i<g_obj_len; i++)
{
    var a_polygon = new Array( );
    /*
     * Create polygons to form 3D objects in the scene and keep
     * them in the array.
     */

    /*
     * Arrange the rendering order according to the polygons'
     * depths. start and end are starting and ending indices
     * of the group of the polygons of the current object.
     */
    quicksortDepth(a_polygon, start, end);
}

/*
 * Separate polygons in to two groups, a visible and invisible
 * group.
 */
var countVisible = 0; var countInvisible = 0;
var visiblePolygons = new Array( );
var invisiblePolygons = new Array( );
for(var k=0; k<a_polygon_len; k++)
{
    /* Check if this polygon is visible. */
    if(a_polygon[k].isVisible())
        visiblePolygons[countVisible++] = a_polygon[k];
    else
        invisiblePolygons[countInvisible++] = a_polygon[k];
}

```

```

/* Draw invisible polygons before visible polygons. */
var invi_len = invisiblePolygons.length;
var vi_len = visiblePolygons.length;
var total = invi_len + vi_len;
// Render invisible polygons in the scene
for(var i=0; i<invi_len; i++)
{
    /*
    * Use DOM API to clear data in SVG <polygon> tags.
    * <polygon> tags are dynamically created and inserted
    * into this document by the Primitive.createTag(
    * )method.
    */
    g_polygonSVGTag[i].setAttribute("points", "0,0");
    g_polygonSVGTag[i].setAttribute("fill", "none");
    g_polygonSVGTag[i].setAttribute("stroke", "none");
    /*
    * USE DOM to supply projections to the points attribute.
    * These projections are for specifying the polygon area.
    */
    // Get projections in a string format.
    var points = invisiblePolygons[i].stringOfPoints;
    polygonSVGTag[i].setAttribute("points", points);
    /*
    * Apply shading on a polygon surface. The url is
    * referred to an associated <linearGradient> tag so that
    * it can shade this polygon surface correctly. We use
    * a polygon's id and an object's id to identify a
    * correct url.<linearGradient> tags are first
    * dynamically created and inserted into the document by
    * Primitive.createTag( ).
    */
    var url = "url(#" + invisiblePolygons[i].object.id +
        invisiblePolygons[i].id + ")";
    polygonSVGTag[i].setAttribute("fill", url);
}
// Render visible polygons in the scene
for(var i=0; i<vi_len; i++)
{
    /* Reset <polygon> tags. */
    polygonSVGTag[i + invi_len].setAttribute("points",
        "0,0");
    polygonSVGTag[i + invi_len].setAttribute("fill",
        "none");
    polygonSVGTag[i + invi_len].setAttribute("stroke",
        "none");
    var projection = visiblePolygons[i].vertices;
    var points = visiblePolygons[i].stringOfPoints;
    polygonSVGTag[i + invi_len].setAttribute("points",
        points);

    /* Apply shading on the polygon surface. */
    var url = "url(#" + visiblePolygons[i].object.id +

```

```
        visiblePolygons[i].id + " ");
    polygonSVGTag[i + invi_len].setAttribute("fill", url);
}

/* Reset the perspective matrix to be an identity matrix. */
perspective.reset( );
}
```



## 5. Optimization

### 5.1 Optimization of The Graphics-Rendering Algorithms

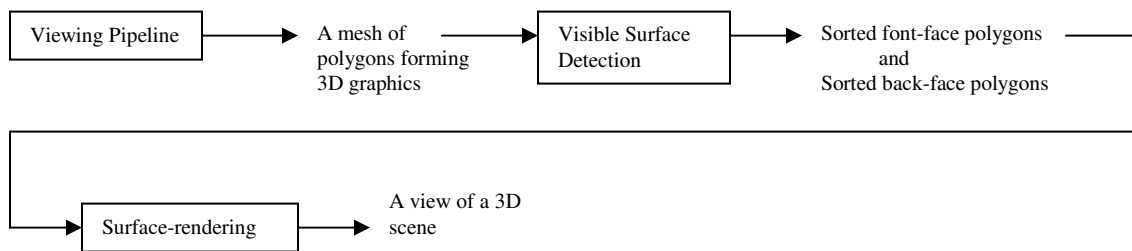
We have developed three versions of the translator in this project.

#### The first version

In this version, we apply the algorithms as explained in section 4 in a straightforward manner to render a view of 3D graphics on the screen. This non-optimized version draws every single polygon of the scene whether or not it is visible or invisible from a viewing position.

#### The second version

There are three general steps to generate a view of a 3D scene, the viewing pipeline step, the visible surface detection step, and the surface-rendering step. A diagram below shows the general procedures in our translator application.



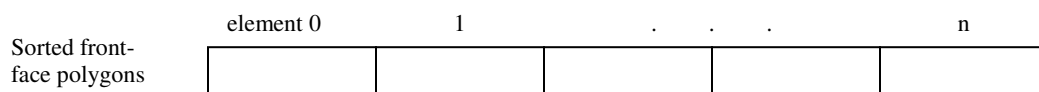
**Figure 5-1** General procedure used in this project to produce a view of a 3D scene.

After the program has been analyzed thoroughly, the surface-rendering process consumes about 50% of the total system processing time (we have included the performance test of a surface-rendering process in the *deployment* section.). If we can

reduce computational time in this process, we can successfully improve program performance. So, in the second version, we try to optimize our program by selectively rendering only polygons that are visible from the viewing point; invisible polygons are not rendered on the screen as they are in the first version. The second version reduced the processing time of this cpu-intensive process by almost 50% and improved the system performance by 30-40%. We ran several test cases to observe the performance of all three versions. This performance can be found in the *deployment* section.

### The third version

In this version, we apply a hidden-surface removal algorithm to remove all the occluded front-face polygons. Since some front-face polygons might be hidden by other front-face polygons in the scene, by detecting which ones are hidden polygons, we can reduce the number of front-face polygons rendered in the scene. Consequently, we can reduce the calculation time in the surface-rendering procedure, which is the most expensive process. Referring back to the general procedures for rendering 3D scenes in the previous section, once the visible-surface-detection process is ended, we get an array of sorted front-face polygons. The array is arranged from the deepest polygon to the closet polygon. We want to reduce the number of polygons in this array by removing the hidden surfaces.

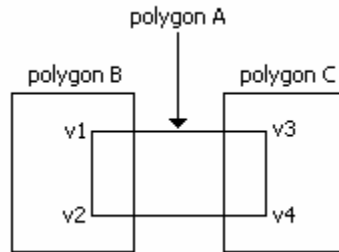


**Figure 5-2** An array of sorted front-face polygons.

Element 0 is the deepest polygon and rendered first; element  $n$  is the closest polygon and rendered last. Obviously, polygon  $n$  is not occluded by any other polygons in the scene,

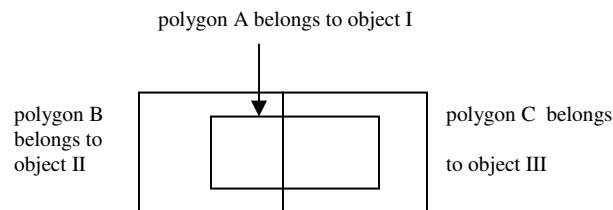
so we do not need to check this one. When we check a polygon to see if it is a hidden polygon, we only check against the closer polygons. For example, this operation starts checking from polygon  $n-1$  back to polygon 0. We check polygon  $n-1$  with polygon  $n$ , polygon  $n-2$  with polygons  $n-1$  and  $n$ , polygons  $n-3$  with polygons  $n-2$ ,  $n-1$ , and  $n$ , and so on. We can avoid checking all possible closer polygons; for instance, we can check only closer polygons that belong to different objects because front-face polygons of the same object will not hide each other.

The process of checking hidden surfaces is not as cheap as we expected. When we check each pair, it is not a single operation as thought. Each polygon, in our application, is composed of three, four, or eight vertices. The algorithms we use to determine if polygon A is hidden is to check if every vertex of polygon A is occluded by any front-face polygons. If so, we can determine that polygon A is occluded by some polygons, and can be removed. There are some cases that every vertex of polygon A is hidden by polygons but the surface is not a hidden surface that we can remove. For example,



**Figure 5-3** Polygon A cannot be considered as a hidden surface.

All vertices of polygon A are hidden by polygon B and C but polygon A cannot be considered as a hidden surface. In this example, polygon A, B, and C definitely belong to different objects in the scene. To handle this case, if a vertex is occluded by a polygon, we assign an object's ID to a vertex. The object's ID is the ID of the object the polygon belongs to. In this example, polygon B belongs to a sphere whose ID is 5 and polygon C belongs to a cone whose ID is 6. We assign 5 to v1 and v2 and 6 to v4 and v5. We can determine polygon A is occluded by some surfaces in the scene if all vertices of polygon A are hidden by polygons of the same object, that is they all have the same object's ID. However, this approach has some limitations. It cannot detect a polygon which is completely hidden by several polygons that belong to different objects. For example,



**Figure 5-4** Fail to detect polygon A as a hidden surface.

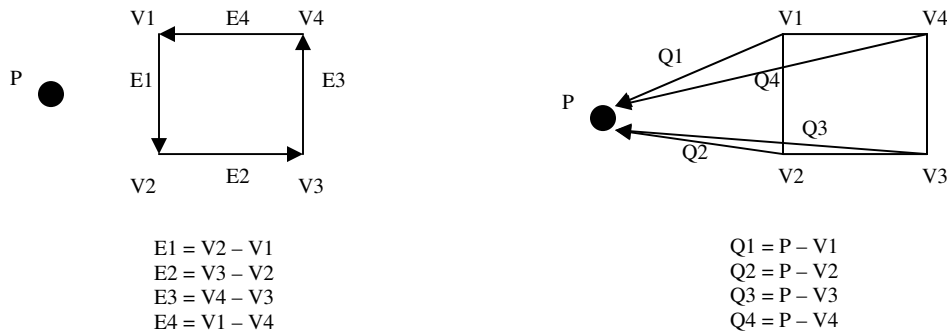
In the above example, it fails to detect polygon A as a hidden surface because not all vertices of polygon A hold the same value and we use this to determine if polygon A is a hidden surface. Even though this approach cannot detect every possible hidden polygon,

it is guaranteed to produce a correct view of a scene. Below is the pseudo code of the hidden-surface removal algorithm.

```
function removeHiddenPolygons(visiblePolygons)
{
    // Check all polygons except for the closet one)
    for(var i=visiblePolygons.length - 2; i=0; i--)
    {
        var polygon_a = visiblePolygons[i];
        var a_id = visibleSurfaces[i].objectID;

        /*
        * Check with other closer polygons which belong to
        * different objects.
        */
        for(var j=visiblePolygons.length - 1; j<i; j--)
        {
            var polygon_b = visiblePolygons[j];
            var b_id = visiblePolygons[j].objectID;
            if( a_id != b_id )
            {
                /*
                Check every vertex of polygon_a if it is hidden by
                polygon_b. if so, assign b_id to that vertex.
                */
            }
        }
        /*
        Determine if polygon_a is a hidden surface if all vertices of
        polygon_a hold the same value.
        */
    }
    /*
    Remove hidden surfaces from the array of visiblePolygons
    */
}
```

We determine if a vertex is occluded by a polygon by checking if the vertex or point is inside a polygon plane. We create vectors between two polygon vertices and vectors between a polygon vertex and the point we want to check.

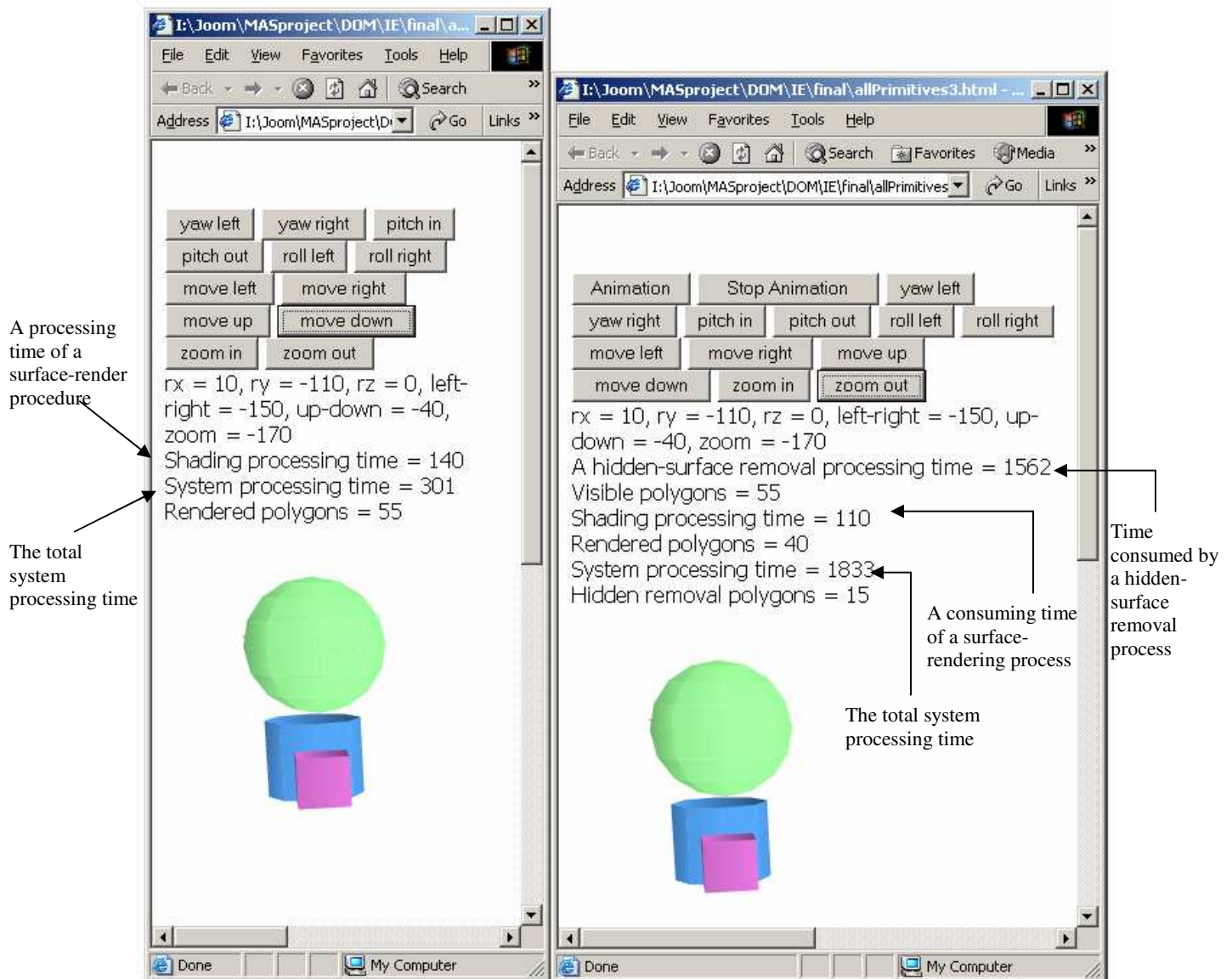


**Figure 5-5** Vectors between two polygon vertices, and vectors between a polygon vertex and a checking point.

Then we find the cross product between vector E1 and Q1, E2 and Q2, and so on (how to find a cross product of two vectors is included in *Appendix A*). If there is a cross product that has a z-value less than zero, then the point is outside the polygon plane or the vertex is not hidden by this polygon surface. This technique can be used with any kinds of convex polygons, triangle, rectangle, pentagon, octagon, etc.

We developed a third version with the hidden-surface removal algorithm that we thought would definitely improve program's performance. Surprisingly, the third version drops performance by over 100 % because we pay so much more for the overhead of the hidden-surface removal procedure than the time we save from the surface-rendering process. The first version, which applies the rendering algorithm in a straightforward manner, still has better performance than the third version. Even if the third version successfully decreases processing time of the surface-rendering procedure, it does not

speed up the program at all. Below shows how much the hidden-surface removal algorithm costs.



**Figure 5-6** Performance comparison between version 2 (left) and version 3 (right).

The processing time of the surface-rendering procedure of version 3 is less than that of version 2's but the total system processing time of version 3 is much more than that of version 2's because of the overhead from the hidden-surface removal process. We include a comparison table of all three versions in the *deployment* section.

Of all three versions, the second version is the best one and has time complexity  $O(n^2 \ln n)$ ;  $n$  is the number of polygons (we include the performance data of the second version in the *deployment* section). Theoretically, the third version would have had better performance and would beat the second version. Ironically, after we implemented it, the third version has worse performance than the other two versions because the hidden-surface removal process is very prohibitive and costs much more than a surface-rendering process.

## **5.2 Performance Hints to Speed Up JavaScript Code**

Another way to boost our program's performance is to improve JavaScript code. In C or C++, a programmer is allowed to manually allocate and deallocate system resources. In JavaScript, every time a program creates a string, array, or object, the interpreter dynamically allocates memory to store that entity. The interpreter reclaims the memory back when it is no longer in use by a garbage collector. Garbage collection is automatic and invisible to the programmer. In JavaScript, we cannot monitor, turn on, or turn off a garbage collector as we can in Java. Even if we cannot directly control and manage resources in JavaScript, here are some hints that help speed up the code:

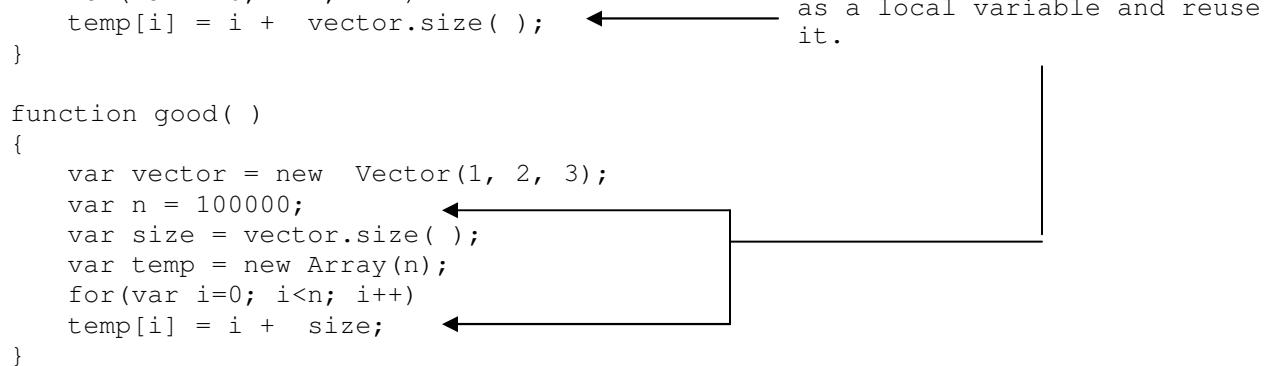


1. Avoid using a dot operator because a dot operator requires time to dereference. The best solution is to locally cache objects if we need to access them very often.

```
function bad( )
{
    var vector = new Vector(1, 2, 3);
    var n = 100000;
    var temp = new Array(n);
    for(var i=0; i<n; i++)
        temp[i] = i + vector.size( );
}

function good( )
{
    var vector = new Vector(1, 2, 3);
    var n = 100000;
    var size = vector.size( );
    var temp = new Array(n);
    for(var i=0; i<n; i++)
        temp[i] = i + size;
}
```

We can avoid repeatedly calling `.size( )` by saving it as a local variable and reuse it.



We have done an experiment to verify this idea; for instance, we ran both the above functions with a different number of  $n$  on the same machine to measure processing time.

<b>n</b>	<b>bad( )'s processing time</b>	<b>good( )'s processing time</b>
1000	10 ms	~0-1 ms
10000	110 ms	31 ms
100000	1192 ms	591 ms
1000000	27870 ms	20820 ms

**Table 5-1** Processing time of test functions for *dot* operator.

Hence, avoiding the use of dot operators can improve the JavaScript code performance.

2. Use var statements when possible. It can speed up your code by allowing the compiler to generate special codes to access these variables. We also ran a few test cases to verify

this statement; below is the sample code and a table showing the run times of these compared functions for values of  $n$ .

```
function bad()
{
    n = 100000;
    temp = new Array(n);
    for(i=0; i<n; i++)
    {
        sqrt = Math.sqrt(i);
        cos = Math.cos(i);
        sin = Math.sin(i);
        tan = Math.tan(i);
        temp[i] = sqrt*cos*sin*tan;
    }
}

function good()
{
    var n = 100000;
    var temp = new Array(n);
    for(var i=0; i<n; i++)
    {
        var sqrt = Math.sqrt(i);
        var cos = Math.cos(i);
        var sin = Math.sin(i);
        var tan = Math.tan(i);
        temp[i] = sqrt*cos*sin*tan;
    }
}
```

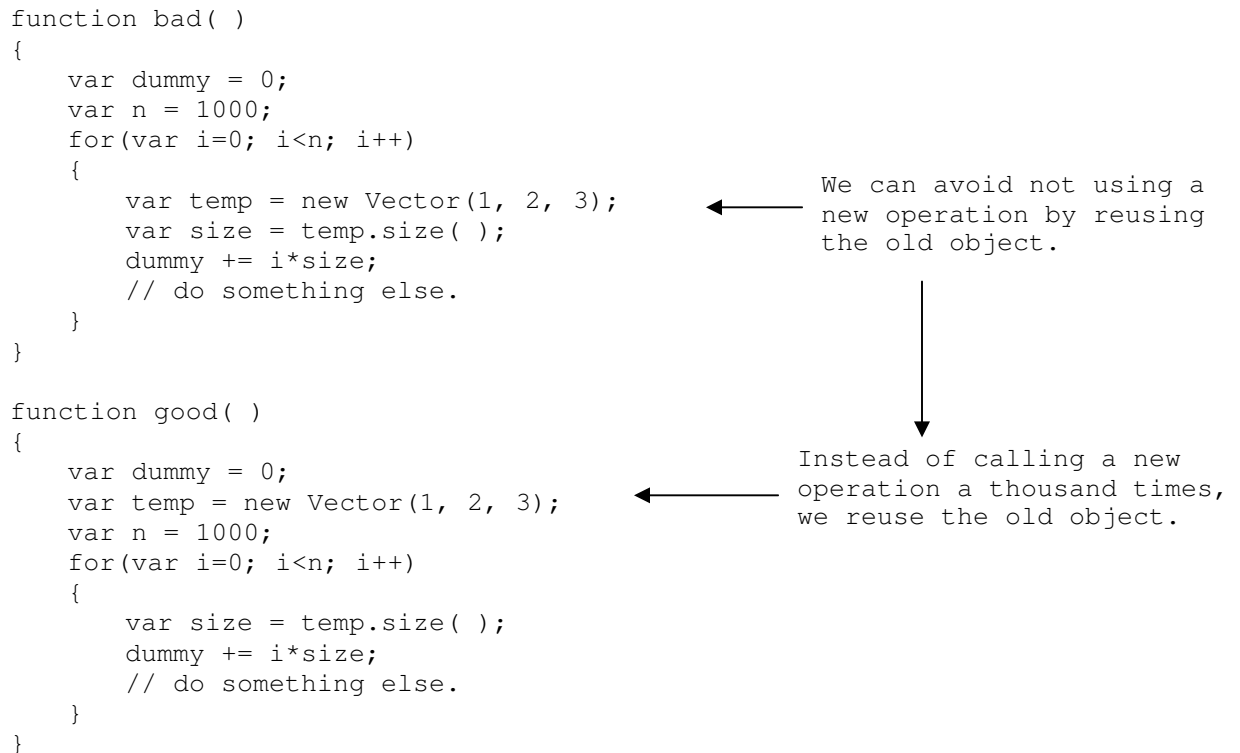
n	bad()’s processing time	good()’s processing time
1000	10 ms	10 ms
10000	140 ms	90 ms
100000	1863 ms	1572 ms
1000000	156405 ms	59235 ms

**Table 5-2** Processing time of test functions for *var* statement.

The processing-time table above shows that using a *var* statement can speed up the JavaScript code.

3. Try to reuse the object when possible because a constructor `new` function can be slow.

For example,



We also tested these functions to verify this claim and the table below displays the results of comparing these two functions.

n	bad( )'s processing time	good( )'s processing time
1000	20 ms	~0-1 ms
10000	200 ms	10 ms
100000	1983 ms	80 ms
1000000	156405 ms	59235 ms

**Table 5-3** Processing time of test functions for *new* operator.

The results from the above table show that reusing the old object can speed up the JavaScript code.

4. It is recommended to use the forms of the array constructor that specify a size or take a list of initial elements. For example,

```
function bad( )
{
    var n = 10000;
    var m = 10;
    var a = new Array();
    for (var i=0; i< n; i++)
    {
        a[i] = new Array();
        for(var j=0; j<m; j++)
            a[i][j] = j;
    }
}

function good( )
{
    var n = 10000;
    var m = 10;
    var a = new Array(n);
    for (var i=0; i< n; i++)
    {
        a[i] = new Array(m);
        for(var j=0; j<m; j++)
            a[i][j] = j;
    }
}
```

The code could be sped up by changing the constructor call to `new Array(n)`. A constructor call like that indicates that an array should be used for the first  $n$  entries of the array. We tested both functions to verify this claim. Based on our experiments as shown in the table below, specifying the array size does not always improve the processing time of a program. After we compared the time between `bad()` and `good()` on four different JavaScript engines, the IE, Mozilla-browser, Netscape, and Rhino JavaScript engine, the results show that this statement is not always true. We tested three different  $n$  and we ran five times for each  $n$  in this experiment. In IE, Mozilla, and Netscape browsers, usually `bad()` takes less time than `good()`; however, sometimes `good()` is

better than `bad()`. In Rhino, when  $n$  is small like  $n=100$ , sometimes `good()` has a better performance than `bad()` but not always. When  $n$  is a big number such as  $n=1000$ , or  $10000$  or  $50000$ , `good()` always has a better processing time than `bad()`.

Below is a table showing test results we got from four different engines.

n	IE		Mozilla		Netscape		Rhino	
	bad()	good()	bad()	good()	bad()	good()	bad()	good()
100(1)	1	10	1	10	10	1	10	10
100(2)	1	10	10	1	10	10	10	10
100(3)	10	1	1	10	10	1	10	20
100(4)	10	1	10	1	1	1	11	20
100(5)	1	10	1	10	10	10	21	10
1000(1)	40	50	40	90	260	231	160	141
1000(2)	50	50	50	120	230	401	160	150
1000(3)	60	50	50	100	91	290	160	140
1000(4)	50	60	40	100	80	200	240	140
1000(5)	60	60	50	90	90	210	150	140
10000(1)	951	1392	801	641	1733	2073	1612	1462
10000(2)	1291	1402	861	651	1723	1772	1712	1462
10000(3)	1252	1392	841	641	1973	1752	1743	1572
10000(4)	1271	1392	891	661	1862	1733	1622	1582
10000(5)	1282	1392	671	701	2163	1762	1632	1553
50000(1)	15803	16523	3375	3525	6640	7451	9744	7471
50000(2)	15773	16323	3225	4526	6800	7310	8472	7571
50000(3)	15562	16334	3344	3275	6680	7501	7931	7721
50000(4)	15462	16294	3275	3395	6710	7581	8662	7091
50000(5)	15622	16074	3365	3435	6699	7601	8102	7190

**Table 5-4** Processing time on four different Java engines.

From the experiment, we can conclude that this claim is not true for the IE, Mozilla, and Netscape JavaScript engines. It is true in the Rhino JavaScript engine, when  $n$  is a large number.

Unfortunately, there is no way to turn on or off the garbage collector in JavaScript as we can do in Java, so that we might be able to monitor how the interpreter manages resources differently between specified and unspecified arrays.

In this project, we apply these hints to enhance the performance in JavaScript code. Objects, strings, arrays, and variables in this program are created using a var statement. Usually, we locally cache the objects when we need to access them very often as well as try to reuse the old objects as much as possible.

## 6. Deployment

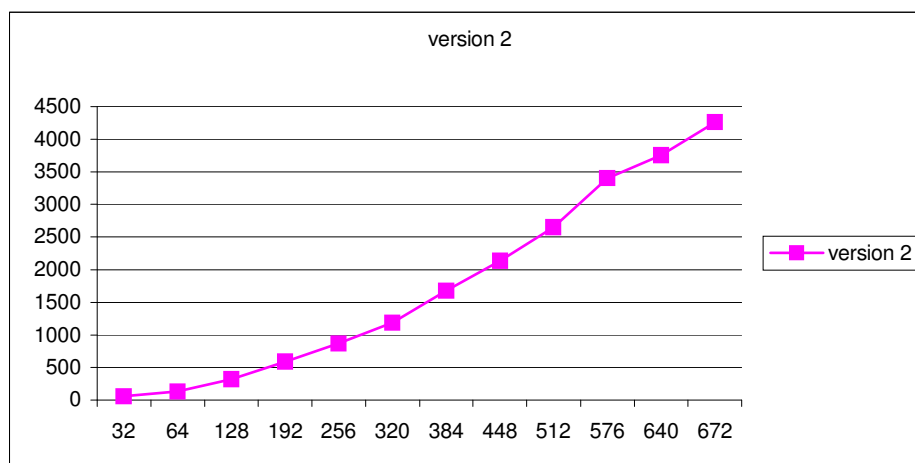
Each test case that we ran to observe the performance of our program was tested on a machine with 1400 MHz Pentium 4 CPU with 52 MB RAM.

### 6.1 Performance Observations on The Second Version of The Translator of Our Program

We have observed performance of the translator version 2, which has time complexity  $O(n^2 \ln n)$ ;  $n$  is a number of polygons. Below is a table showing processing time of different test cases that we experimented with our translator version 2 and a graph of the result. The graph shows the behavior of the application that conforms to the time complexity we analyze.

# of Polygons	Processing Time (milliseconds)
32 (a cone)	~30-60
64 (a sphere)	~110-130
128 (two spheres)	~290-320
192 (three spheres)	~560-590
256 (four spheres)	~750-870
320 (five spheres)	~1090-1190
384 (six spheres)	~1590-1680
448 (seven spheres)	~1972-2133
512(eight spheres)	~2444-2654
576(nine spheres)	~3024-3405
640(ten spheres)	~3225-3755
672(ten spheres + one cone)	~3625-4266

A processing-time table of different test cases running on the translator version 2



A graph associated with the results displayed in the above table

## 6.2 Performance Observations Among Three Versions of The Translator

We ran several test cases to observe the performance of versions 1, 2 and 3. Below are some comparison performance tables among these three versions. The second version of our program is the fastest one.

Test Case I: 36 polygons

Version	Processing Time	% Performance Increases or Drops (compared with Version I)
I	~90-110 ms	-
II	~60-80 ms	Increases 20-30 %
III	~230-270 ms	Drops > 100%

Test Case II: 113 polygons

Version	Processing Time	% Performance Increases or Drops (compared with Version I)
I	~ 410-430 ms	-
II	~ 240-290 ms	Increases 30-45 %
III	~ 680-2654 ms	Drops > 70 %

Test Case III: 192 polygons

Version	Processing Time	% Performance Increases or Drops (compared with Version I)
I	~ 830-850 ms	-
II	~ 560-590 ms	Increases 32-36 %
III	~ 3895-6690 ms	Drops > 100 %

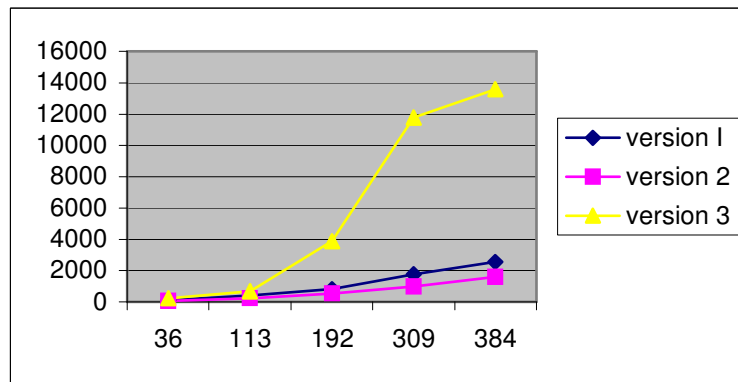


Test Case IV: 309 polygons

Version	Processing Time	% Performance Increases or Drops (compared with Version I)
I	~ 1780-1800 ms	-
II	~ 1000-1400 ms	Increases 30-40 %
III	~ 11767-42170 ms	Drops down > 100 %

Test Case V: 384 polygons

Version	Processing Time	% Performance Increases or Drops (compared with Version I)
I	~ 2550-2600 ms	-
II	~ 1590-1680 ms	Increases 33-37 %
III	~ 13580-44074 ms	Drops > 100 %



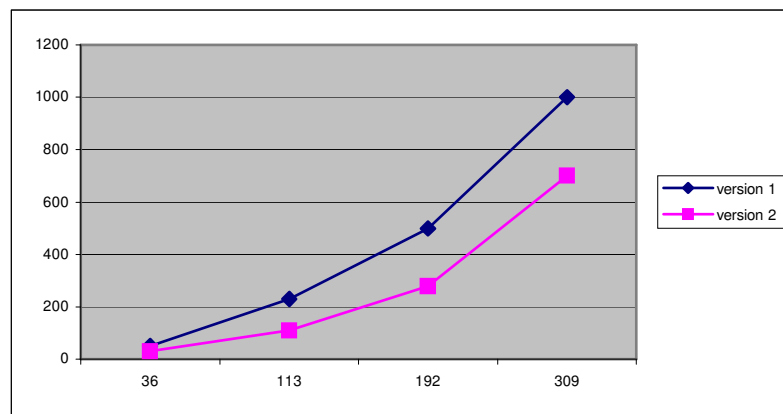
A graph associated with the results displayed in the above table

### 6.3 Performance Observations on The Surface-Rendering Process of The Translator Versions 1 and 2

We ran four test cases to observe the surface-rendering process between the translator versions 1 and 2. The second version of the translator can reduce the processing time of this process by almost 50 %

# of Polygons	Surface-rendering processing time in version 2	Surface-rendering processing time in version 2
36	~50-80 ms	~ 30-40 ms
113	~230-320 ms	~ 110-140 ms
192	~500-690 ms	~ 280-310 ms
309	~1000-1770 ms	~ 700-930 ms

The processing time of the surface-rendering process between version 1 and 2 of our translator



A graph associated with the results displayed in the above table

#### 6.4 Processing Time of A Surface-Rendering Process

We tested our translator version 2 with a different number of polygons in the scene to observe a process time of the surface-rendering process. We found that it consumes about 50% of the total application processing time.

# of Polygons	System processing time	Surface-rendering processing time
36	~70-80 ms	~ 30-40 ms
113	~ 240-290 ms	~ 110-140 ms
192	~ 560-590 ms	~ 280-310 ms
309	~ 1000-1400 ms	~ 700-930 ms

A processing-time table of a surface-rendering procedure

## 7. Conclusion

### 7.1 Project Achievements

Originally, SVG was designed to display 2D line-art graphics, not to render 3D graphics. SVG is a fixed simple XML application, not a robust graphics package like OpenGL. Using SVG to produce realistic 3D graphics, we have to develop a program for rendering the graphics from scratch. Thus we faced many challenges to use this tool to display realistic 3D objects. Several rendering algorithms in computer graphics cannot be directly applied in this application. One of the most important problems we faced was how to render a mesh of polygons in a correct order. Many computer graphics algorithms use the image-space method to describe such graphics. Unfortunately, we cannot apply this approach to our application. We invented our own technique to solve this problem. We sampled the middle point of a polygon to set the depth of this polygon and use this depth to arrange the rendering order. This approach is very simple and fast requiring only a small number of calculations.

We successfully delivered an application that translates an X3D document to an SVG document. The X3DToSVG application can produce a correct view of smooth shaded 3D objects and provide features that allow the user to view a scene in different perspectives. We successfully support `<Appearance>`, `<Group>`, `<Transform>`, `<Box>`, `<Cone>`, `<Cylinder>`, and `<Sphere>` tags of the source X3D document.

We tried to optimize our translator and we successfully improved the performance by 30-40%. Besides trying to optimize the rendering-graphics algorithm, we want to speed up JavaScript code as much as we can. We learn that the interpreter dynamically manages all resources in JavaScript environment. So we are not allowed to control and manage memory resources manually as we can in other programming languages such as C and C++. We try to search for hints and techniques that might speed up the code. We found four techniques which are claimed that they to boost the JavaScript code. We ran several test cases to verify these claims and discovered that not all of them can speed up the code as they claimed. More details about this can be found in the *optimization* section.

Usually, JavaScript has been used for small-scale programs such as web programming to interact with the user, control the browser, validate a document, and dynamically create HTML content. A graphics-rendering program is not a typical task implemented in JavaScript. It is challenging to attempt to use this tool for this nontrivial task. We have successfully used JavaScript to develop a sophisticated and nontrivial program for rendering 3D graphics.

## **7.2 Future Work**

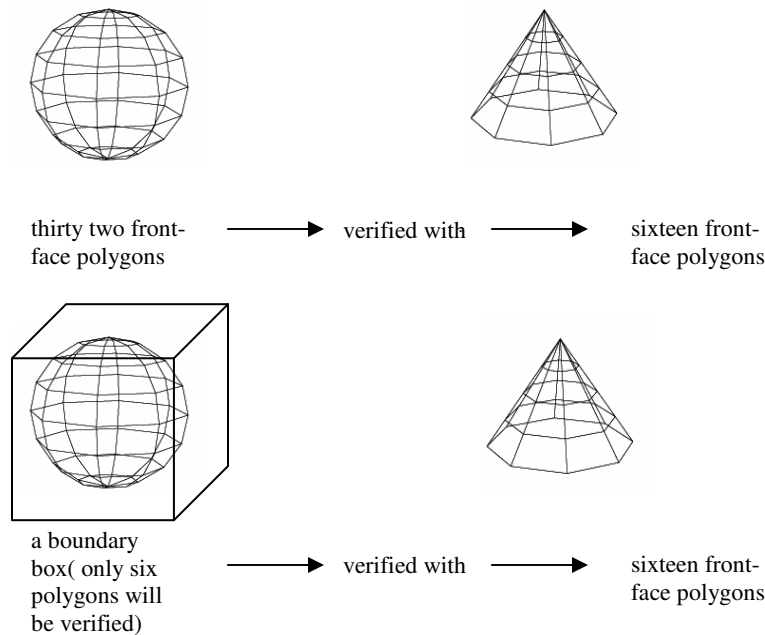
As mentioned in the *limitation* section, so far our application cannot guarantee to display overlapping or intersecting objects correctly. Since the algorithm that we use in our application right now displays polygons in the scene as a whole and the SVG engine does not support internally describing a polygon at a pixel level, we cannot display a partial

area of a polygon. To overcome this problem, we should be able to subdivide a polygon so that our translator can partially display polygons. However, we did not implement a process of subdividing polygons. We leave this topic for future work.

Now, if there is any part of the scene getting outside the viewing limit, that part is still rendered by our translator and does not get clipped. We waste time rendering the part which will not be seen. A clipping operation is another possible future work for this project. Performing clipping operations may speed up the program because theoretically we save time for not rendering the unseen part that is outside a viewing space; however, it might not improve the program performance if the overhead time of the clipping process is more than the time we save from the rendering process. It may or may not end up like the hidden-surface removal algorithm in the third version.

The hidden-surface removal algorithm that we apply in the third version of our translator program is so expensive that the performance of this version is far worse than the other two versions. A number of front-face polygons to be checked by a hidden-surface removal process directly affect the computational time. If we can reduce this number, the time of this process should decrease. We have an idea to optimize this procedure that may contribute to a better performance. Now, we check every front-face polygon in the scene with other closer polygons of different objects. We want to reduce the number of polygons to be checked. Instead of performing a hidden-surface removal task, we want to do a **hidden-object removal** task. We will check if a whole object is hidden by any

front-face polygons in the scene. We create a boundary box wrapping an object in the scene. The boundary box, which composes of six polygons like a common box, is used to represent the object that it wraps. We check these six polygons with the other closer polygons belonging to different objects. If all of them are hidden by some front-face polygons, we can remove the whole object from the scene. This operation can reduce the number of polygons to be checked; for example, suppose a sphere in the scene has thirty-two front-face polygons that should be checked with the sixteen front-face polygons of a cone. Using the old approach, these thirty-two polygons of a sphere are required to be compared with the sixteen polygons of a cone. If we use a boundary box representing a sphere, we can reduce the number of polygons from thirty-two to six.



However, the new approach processes at an object level, not at a polygon level. The time of a surface-rendering process will not decrease if an object is partially hidden by some

polygons. We have not implemented this approach yet. It may or may not significantly improve the performance. We leave this for future work.

## References

- [ANM97] Andreal L. Ames, David R. Nadeau, John L. MoreLand. VRML2.0 Sourcebook 2<sup>nd</sup> ed. Wiley Computer Publishing. 1997.
- [B02] Norris Boyd. Rhino: JavaScript for Java. The Mozilla Organization. 2002.
- [BH97] Donald Hearn, M. Pauline Baker. Computer Graphics C version 2<sup>nd</sup> ed. Prentice Hall. 1997.
- [ECMA99] Standard ECMA-262 ECMAScript Language Specification 3rd ed. <http://www.ecma.ch/ecma1/stand/ecma-262.htm>. ECMA. 1999.
- [DOM1] Document Object Model (DOM) Level 1 Specification. <http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001>. W3C
- [DOM2] Document Object Model (DOM) Level 2 Specification. <http://www.w3.org/TR/1999/CR-DOM-Level-1-19991210>. W3C
- [E02] J David Eisenberg. SVG Essential. O'Reilly. 2002.
- [F02] David Flanagan. JavaScript The Definitive Guide 4<sup>th</sup> ed. O'Reilly. 2002.
- [3D01] 3D Geometry <http://www.kevlindev.com/geometry/3D/>. Oct 31, 2002.
- [JOI01] JavaScript Object Inheritance <http://www.kevlindev.com/tutorials/javascript/inheritance/>. Oct 31, 2002.
- [MOZ00] Mozilla Rhino: JavaScript for Java: Performance Hints <http://www.mozilla.org/rhino/perf.html>. Oct 2, 2000.
- [MOZ02] Mozilla SVG Project <http://www.mozilla.org/projects/svg/>. The Mozilla Organization.
- [W3C00] The Extensible HyperText Markup Language(XHTML) 2<sup>nd</sup> ed. <http://www.w3.org/TR/2002/REC-xhtml1-20020801>. W3C 2002
- [W3C01] Scalable Vector Graphics (SVG) Specification 1.0. <http://www.w3.org/TR/SVG/>. W3C.

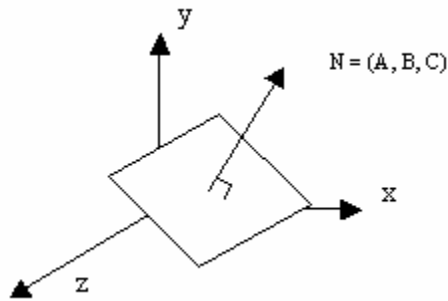


[W3DC01] Extensible 3D (X3D) Graphics Working Group.  
<http://www.web3d.org/x3d.html>. Web 3D Consortium. 2001.

## Appendix A

### A vector normal to a polygon surface

A vector, normal to the surface of a plane described by the equation  $Ax + By + Cz + D=0$ , has Cartesian components (A, B, C). Here (x, y, z) is any point on the plane, and the coefficients A, B, C, and D are constants describing the spatial properties of the plane.



A vector N normal to a plane surface

We can obtain the values of A, B, C, and D by solving a set of three plane equations using the coordinate three successive polygon vertices,  $(x_1, y_1, z_1)$ ,  $(x_2, y_2, z_2)$ , and  $(x_3, y_3, z_3)$ , and solve the following set of simultaneous linear plane equations for the ratios  $A/D$ ,  $B/D$ , and  $C/D$  and  $D \neq 0$ :

$$(A/D) x_k + (B/D) y_k + (C/D) z_k = -1, \quad k = 1, 2, 3$$

The solution of this set of equations can be obtained in determinant form using Cramer's rule. Using this formulation we can calculate coefficients in the form.

$$A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2)$$

$$B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2)$$

$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

$$D = -x_1(y_2 z_3 - y_3 z_2) + x_2(y_3 z_1 - y_1 z_3) + x_3(y_1 z_2 - y_2 z_1)$$

### Cross product of two vectors

Multiplication of two vectors to produce another vector is defined as

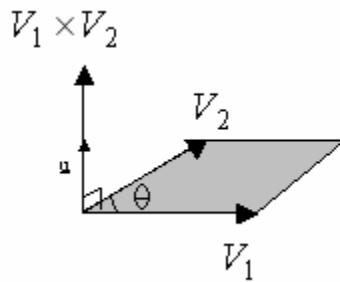
$$V_1 \times V_2 = u |V_1| |V_2| \sin \theta, \quad 0 \leq \theta \leq \pi$$

$u$  is a unit vector of the cross product vector  $V_1 \times V_2$  and  $\theta$  is the angle between  $V_1$  and  $V_2$ .

We can also express the cross product in terms of vector components in a specific reference frame. In the Cartesian coordinate system, we calculate the components of the cross product as

$$V_1 \times V_2 = (V_{1y}V_{2z} - V_{1z}V_{2y}, V_{1z}V_{2x} - V_{1x}V_{2z}, V_{1x}V_{2y} - V_{1y}V_{2x})$$

The cross product of two vectors is a vector in a direction perpendicular to the two original vectors and with a magnitude equal to the area of the shaded parallelogram.



The cross product of two vectors

## Appendix B

### X3DToSVG.dtd

```
<!ENTITY % GeometryNodes" ( Box | Cone | Cylinder | Sphere | Text ) " >
<!ENTITY % GroupingNodes      " Group | Transform " >
<!ENTITY % SceneLeafNodes     " Shape " >
<!ENTITY % SceneNodes         " (Shape)*, ( %GroupingNodes;)*" >
<!ENTITY % ChildrenNodes      " %GroupingNodes; | %SceneLeafNodes; " >
<!ENTITY % Children           "( %ChildrenNodes; )* " >

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

<!ELEMENT Appearance (Material) >

<!ELEMENT Box EMPTY>
<!ATTLIST Box
    size CDATA "10 10 10">

<!ELEMENT Cone EMPTY>
<!ATTLIST Cone
    bottomRadius CDATA "10"
    height CDATA "20">

<!ELEMENTCylinder EMPTY>
<!ATTLIST Cylinder
    height CDATA "20"
    radius CDATA "10">

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

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

<!ELEMENT Sphere EMPTY >
<!ATTLIST Sphere
    radius CDATA "10">

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

<!ELEMENT Material EMPTY >
```

```
<!ATTLIST Material
    diffuseColor      CDATA "0.8 0.8 0.8">
```

## AllPrimitives.xml

This is a sample X3D document. It is an input file for AllPrimitives2.html.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE X3D SYSTEM "X3DToSVG.dtd">
<X3D>
  <Scene>
    <Transform translation="-20 -10 0">
      <Shape>
        <Appearance>
          <Material diffuseColor="0.596 0.984 0.596"/>
        </Appearance>
        <Sphere radius="20"/>
      </Shape>
    </Transform>

    <Transform translation="30 -10 0">
      <Shape>
        <Appearance>
          <Material diffuseColor="1 0.388 0.278"/>
        </Appearance>
        <Cone bottomRadius="20" height="20"/>
      </Shape>
    </Transform>

    <Transform translation="-20 30 0">
      <Shape>
        <Appearance>
          <Material diffuseColor="0.854 0.439 0.839"/>
        </Appearance>
        <Box size="15 15 15"/>
      </Shape>
    </Transform>

    <Transform translation="30 30 0">
      <Shape>
        <Cylinder radius="15" height="25"/>
      </Shape>
    </Transform>
  </Scene>
</X3D>
```

## AllPrimitives2.html

This is source code for the translator version 2. This file requires AllPrimitives.xml as an input document.

```
<!--
#File Name: allPrimitives2.html
#Purpose: It is to demonstrate the second version of our translator.
#It will show a view of a box, cone, cylinder, and sphere as described
#in a provided X3D source file.
#Usage: To run this document, it requires IE 6.XX with an SVG plugin.
#It needs an X3D file allPrimitives.xml together with X3DToSVG.dtd.
#Both of these files are already included in Appendix B as well.
#Class: CS298 writing project.
#Programmer: Paungkaew Sangtrakulcharoen
#
-->
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:svg="http://www.w3.org/2000/svg">

  <!--
  # The next four lines of code are Microsoft propriety methods of
  # emulation XML namespaces. We do not need this code, if we are
  # using a Mozilla-SVG browser.
  -->
  <object id="AdobeSVG"
        classid="clsid:78156a80-c6a1-4bbf-8e6a-3cd390eeb4e2">
  </object>

  <?import namespace="svg" implementation="#AdobeSVG"?>

  <body id="start" onload="buildSVG()">
    <p>
      <!-- Create buttons for the user to change
      perspectives. -->
      <input type="button" id="btnYawLeft" value="yaw
        left" onclick="yawLeft()" />
      <input type="button" id="btnYawRight" value="yaw
        right" onclick="yawRight()" />
      <input type="button" id="btnPitchIN" value="pitch
        in" onclick="pitchIn()" />
      <input type="button" id="btnPitchOut"
        value="pitch out" onclick="pitchOut()" />
      <input type="button" id="btnRollLeft" value="roll
        left" onclick="rollLeft()" />
      <input type="button" id="btnRollRight"
        value="roll right" onclick="rollRight()" />
      <input type="button" id="btnMoveLeft" value="move
        left" onclick="moveLeft()" />
      <input type="button" id="btnMoveRight"
        value="move right" onclick="moveRight()" />
```

```

        <input type="button" id="btnMoveUp" value="move
            up" onclick="moveUp()" />
        <input type="button" id="btnMoveDown" value="move
            down" onclick="moveDown()" />
        <input type="button" id="btnZoomIn" value="zoom
            in" onclick="zoomIn()" />
        <input type="button" id="btnZoomOut" value="zoom
            out" onclick="zoomOut()" />
    </p>
</body>
<script type="text/javascript">
    /* Use the DOM 2 technique, if it is supported. */
    if(document.implementation &&
        document.implementation.createDocument)
    {
        var x3dDoc =
            document.implementation.createDocument("", "", null);
        x3dDoc.async = "false";
        x3dDoc.load("allPrimitives.xml");
    }
    /* Otherwise, use Microsoft proprietary API for IE */
    else if(window.ActiveXObject)
    {
        var x3dDoc = new ActiveXObject("Microsoft.XMLDOM");
        /*
         * This is to assure that the parser halts the
         * execution until the source file is fully load.
         */
        x3dDoc.async = "false";
        /* Load a source X3D file. */
        x3dDoc.load("allPrimitives.xml");
    }

    /* Get the <Scene> node of the source document. */
    var nodes = x3dDoc.getElementsByTagName("Scene");
    var scene = nodes[0];
    /* Clone the <Scene> node including its descendants. */
    var scenel = cloneElement(scene);

    /* Get all the shape nodes from a cloned structure. */
    var g_shapeNodes = scenel.getElementsByTagName("Shape");
    /* Get all the group nodes from a cloned structure. */
    var g_groupNodes = scenel.getElementsByTagName("Group");

    //The "objects" array keeps all objects created this scene.
    var g_objects = new Array(g_shapeNodes.length);

    // The "polygonSVGTag" array stores SVG <polygon> tags.
    var g_polygonSVGTag = new Array();

    /*
     * The "countTag" array is to keep track of a number of
     * SVG <polygon> tags.
     */

```

```

var g_countTag = 0;

/*
 * Get an HTML <body> node of this document where we will
 * insert SVG tags.
 */
var g_bodyNode = document.getElementById("start");

/* Create an SVG root node. */
var g_svg = svg_builder();

/*
 * Whenever perspective buttons- yaw left-right, pitch
 * in-out, roll left-right, are clicked, the angle will
 * be changed by 10 degrees in that perspective.
 */
var g_angle = 10;

/*
 * Whenever zoom in-out and move left-right buttons are
 * clicked, the distance will be changed by 10 units.
 */
var g_distance = 10;

/*
 * The distances in the x, y, and z axis of a current
 * perspective.
 */
var g_tx = 0;
var g_ty = 0;
var g_tz = 0;

/*
 * The angles related to the x, y, z axis of a current
 * perspective.
 */
var g_rx = 0;
var g_ry = 0;
var g_rz = 0;

/*
 * A viewpoint is where we stand and look at the object.
 * In this case, we design the viewing point to be (0, 0,
 * 300) on the positive z axis.
 */
var g_viewPoint = 300;

//The transformation matrix used in viewing transformation

var perspective = new Transform();

var NO_HUE = -1;
var invisible = 0;
var visible = 1;

```



```

/*
 * This function is to create a root element of an SVG
 * document and its other associated attribute nodes.
 * @return a root of the SVG specification.
 */
function svg_builder()
{
    /* Create an "svg" element. */
    var svg = document.createElement("svg:svg");

    /* Create attributes of this element. */
    var width = document.createAttribute("width");
    var height = document.createAttribute("height");
    var viewBox = document.createAttribute("viewBox");

    /*
     * Assign a default value to a width, height and
     * viewBox attribute.
     */
    width.value = "20cm";
    height.value = "20cm";
    viewBox.value = "-100 -50 150 200";

    /* Bind the attributes to the element. */
    svg.attributes.setNamedItem(width);
    svg.attributes.setNamedItem(height);
    svg.attributes.setNamedItem(viewBox);

    return svg;
}

/*****
 * event handling methods
 */
/* These methods will be called when one of buttons is clicked */
/*****
/*
 * This function is called when perspectives have been
 * changed around the y-axis.
 */
function yawLeft()
{
    g_ry += +g_angle;
    drawObjects();
}

function yawRight()
{
    g_ry += -g_angle;
    drawObjects();
}

/*
 * This function is called when perspectives have been
 * changed around the x-axis.

```

```

*/
function pitchIn()
{
    g_rx += g_angle;
    drawObjects();
}

function pitchOut()
{
    g_rx += -g_angle;
    drawObjects();
}

/*
* This function is called when perspectives have been
* changed around the z-axis.
*/
function rollLeft()
{
    g_rz += -g_angle;
    drawObjects();
}

function rollRight()
{
    g_rz += g_angle;
    drawObjects();
}

/*
* This function is called when perspectives is changed
* in a horizontal direction.
*/
function moveLeft()
{
    g_tx += g_distance;
    drawObjects();
}

function moveRight()
{
    g_tx += -g_distance;
    drawObjects();
}

/*
* This function is called when perspectives is changed
* in a vertical direction.
*/
function moveUp()
{
    g_ty += g_distance;
    drawObjects();
}

```

```

function moveDown()
{
    g_ty += -g_distance;
    drawObjects();
}

// This function is called when users request zoom in&out.
function zoomIn()
{
    g_tz += g_distance;
    drawObjects();
}

function zoomOut()
{
    g_tz += -g_distance;
    drawObjects();
}

/*****
/*                                quicksort functions                                */
*****/
/*
* This function is to arrange polygons according to
* their depth.They will be arranged from deepest to
* narrowest.
* @para obj is an array of polygon which we want to
* sort.
* @para from is an index where we want to start sorting.
* @para to is an index where we want to stop sorting
*/
function quicksortDepth(obj, from, to)
{
    if(from < to)
    {
        var s = splitDepth(obj, from, to)
        quicksortDepth(obj,from, s);
        quicksortDepth(obj,s + 1, to);
    }
}

function splitDepth(obj, from, to)
{
    var middle = getMiddle(from, to);
    var pivot = obj[middle].depth;
    var i = from;
    var j = to;

    while(i < j)
    {
        while(obj[j].depth > pivot)
            j--;
        while(obj[i].depth < pivot)

```

```

        i++;

    if(i < j) /* swap the obj i and j. */
    {
        var temp = obj[i];
        obj[i] = obj[j];
        obj[j] = temp;
        i++;
        j--;
    }

    if(i == j)
    {
        if(i != 0 && obj[i].depth < obj[i-1].depth)
        {
            var temp = obj[i-1];
            obj[i-1] = obj[i];
            obj[i] = temp;
        }
        else if(obj[i].depth > obj[i+1].depth)
        {
            var temp = obj[i+1];
            obj[i+1] = obj[i];
            obj[i] = temp;
        }
    }
    return j;
}

/*
 * This function is to arrange objects of this scene
 * according their depth.They will be arranged from
 * deepest to narrowest.
 * @para obj is an array of objects which we want to sort.
 * @para from is an index where we want to start sorting.
 * @para to is an index where we want to stop sorting
 */
function quicksortCenter(obj, from, to)
{
    if(from < to)
    {
        var s = splitCenter(obj, from, to)
        quicksortCenter(obj,from, s);
        quicksortCenter(obj,s + 1, to);
    }
}

function splitCenter(obj, from, to)
{
    var middle = getMiddle(from, to);
    var pivot = obj[middle].center2.z3D;
    var i = from;
    var j = to;

```

```

while(i < j)
{
    while(obj[j].center2.z3D > pivot)
        j--;
    while(obj[i].center2.z3D < pivot)
        i++;

    if(i < j) /* swap the obj i and j. */
    {
        var temp = obj[i];
        obj[i] = obj[j];
        obj[j] = temp;
        i++;
        j--;
    }

    if(i == j)
    {
        if(i != 0 &&
            obj[i].center2.z3D < obj[i-1].center2.z3D)
        {
            var temp = obj[i-1];
            obj[i-1] = obj[i];
            obj[i] = temp;
        }
        else if(obj[i].center2.z3D > obj[i+1].center2.z3D)
        {
            var temp = obj[i+1];
            obj[i+1] = obj[i];
            obj[i] = temp;
        }
    }
}
return j;
}

/*
 * This function is to find the middle number between two
 * integer values.
 * @para x and y are the two integers that we want to
 * find the middle number.
 * @return the middle integer.
 */
function getMiddle(x, y)
{
    if( (x + y)%2) != 0)
        return (x + y - 1)/2;
    else
        return (x + y)/2
}

/*****
/*          hsvTorgb and rgbTohsv function          */

```

```

/*****
/*
 * This function is to convert a color format from hsv to rgb.
 * @para h, s, and v are values ranging from 0 to 1.
 * @return an rgb color.
 */
function hsvTorgb(h, s, v)
{
    var r = 0;
    var g = 0;
    var b = 0;

    if(s == 0) /* Grayscale */
        r = g = b = v;
    else
    {
        if(h == 1)
            h = 0;

        h *= 6.0;
        var i = Math.floor(h);
        var f = h - i;
        var aa = v * (1 - s);
        var bb = v * (1 - (s * f));
        var cc = v * (1 - (s * (1 - f)));

        if(i == 0)
        {
            r = v;
            g = cc;
            b = aa;
        }
        else if(i == 1)
        {
            r = bb;
            g = v;
            b = aa;
        }
        else if(i == 2)
        {
            r = aa;
            g = v;
            b = cc;
        }
        else if(i == 3)
        {
            r = aa;
            g = bb;
            b = v;
        }
        else if(i == 4)
        {
            r = cc;
            g = aa;

```

```

        b = v;
    }
    else if(i == 5)
    {
        r = v;
        g = aa;
        b = bb;
    }
}
var rgb = new Array(r, g, b);
return rgb;
}

/*
 * This function is to convert a color format from rgb to hsv.
 * @para r, g, and b are values ranging from 0 to 1.
 * @return an hsv color.
 */
function rgbTohsv(r, g, b)
{
    var max = Math.max(r, g, b);
    var min = Math.min(r, g, b);
    var delta = max - min;

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

    if(max != 0)
        s = delta/max;
    else
        s = 0;

    if(s == 0)
        h = NO_HUE;
    else
    {
        if(r == max)
            h = (g - b)/delta;
        else if(g == max)
            h = 2 + (b - r)/delta;
        else if(b == max)
            h = 4 + (r - g)/delta;

        h *= 60;
        if(h < 0)
            h += 360;

        h /= 360;
    }
    var hsv = new Array(h, s, v);
    return hsv;
}

```

```

/*****
/*                                     Point3D Class                                     */
*****/
/*
* This class is to represent a 3D point.
* @para x, y, and z are coordinates in x, y, and z axis.
*/
function Point3D(x, y, z) /* Constructor */
{
    this.x3D = x;
    this.y3D = y;
    this.z3D = z;
    this.h = 1;

    // xp, yp, and zp are projection coordinates of this point.
    this.xp = 0;
    this.yp = 0;
    this.zp = 0;
    this.h = 0;

    // By default, we set an object to be invisible.
    this.visibility = invisible;
}

/*
* This method is to set x, y, and z values to a Point3D
* object.
* @para x, y, and z are values we want to set to the
* object.
*/
Point3D.prototype.setXYZ = function(x, y, z)
{
    this.x3D = x;
    this.y3D = y;
    this.z3D = z;
}

/*
* This method is to find viewing coordinates of a
* Point3D object at different perspectives.
* @para matrix is a transformation matrix containing
* rotation angles and translation distances of a current
* perspective in the x, y, and z axis.
*/
Point3D.prototype.rotate_translate = function(matrix)
{
    var x = this.x3D;
    var y = this.y3D;
    var z = this.z3D;

    this.x3D = x*matrix[0] + y*matrix[1] +
                z*matrix[2] + matrix[3];
    this.y3D = x*matrix[4] + y*matrix[5] +
                z*matrix[6] + matrix[7];
}

```



```

        this.z3D = x*matrix[8] + y*matrix[9] +
            z*matrix[10] + matrix[11];
    }

    /*
    * This method is to find new world coordinates of a
    * Point3D object after performing a translation.
    * @para matrix is a translation matrix containing
    * translation distances in the x, y, and z axis.
    */
    Point3D.prototype.translate = function(matrix)
    {
        var x = this.x3D;
        var y = this.y3D;
        var z = this.z3D;

        this.x3D = x + matrix[3];
        this.y3D = y + matrix[7];
        this.z3D = z + matrix[11];
    }

    /*
    * This method is to find new world coordinates of a
    * Point3D object after performing a rotation.
    * @para matrix is a rotation matrix containing rotation
    * angles in the x, y, and z axis.
    */
    Point3D.prototype.rotate = function(matrix)
    {
        var x = this.x3D;
        var y = this.y3D;
        var z = this.z3D;

        this.x3D = x*matrix[0] + y*matrix[1] +
            z*matrix[2];
        this.y3D = x*matrix[4] + y*matrix[5] +
            z*matrix[6];
        this.z3D = x*matrix[8] + y*matrix[9] +
            z*matrix[10];
    }

    /*
    * This method is to find new world coordinates of a
    * Point3D object after performing a scale transformation.
    * @para Sx, Sy, and Sz are scale factors in the x, y,
    * and z axis.
    */
    Point3D.prototype.scale = function(Sx, Sy, Sz)
    {
        var x = this.x3D;
        var y = this.y3D;
        var z = this.z3D;

        this.x3D = x*Sx;

```

```

        this.y3D = y*Sy;
        this.z3D = z*Sz;
    }

    /*
    * This method is to calculate the projected positions of
    * a Point3D object.
    * @para viewingPoint is the viewing position on the
    * positive z axis where we look the object.
    */
    Point3D.prototype.project = function(viewingPoint)
    {
        /
        * xp = x3D * d/(Zprp - z3D), yp = y3D*d/(Zprp - z3D)
        * d = zprp - zvp In this case, the view plane is
        * xy plane, so zvp = 0
        */
        var Zprp = viewingPoint;
        var factor = Zprp/(Zprp - this.z3D);

        this.xp = this.x3D*factor;
        this.yp = this.y3D*factor;
    }

    /*****
    /*          Transform Class                      */
    /*****/
    /* This class is to create a transform matrix. */
    function Transform() /* Constructor */
    {
        this.matrix = new Array(
            1, 0, 0, 0,
            0, 1, 0, 0,
            0, 0, 1, 0,
            0, 0, 0, 1)
    }

    /*
    * This method is to create a translation matrix.
    * @para Tx, Ty, and Tz are translating distances in the
    * x, y, and z axis.
    */
    Transform.prototype.translate = function(Tx, Ty, Tz)
    {
        var matrix = this.matrix;

        matrix[3] = Tx;
        matrix[7] = Ty;
        matrix[11] = Tz;
    }

    /*
    * This method is to create a composition Rz*Rx*Ry
    * transformation matrix for a rotation.

```

```

* @para Rx, Ry, Rz are rotation angles in the x, y, and
* z axis in radian unit
*/
Transform.prototype.rotate = function(Rx, Ry, Rz)
{
    var matrix = this.matrix;

    var cosx = Math.cos(Rx);
    var cosy = Math.cos(Ry);
    var cosz = Math.cos(Rz);
    var sinx = Math.sin(Rx);
    var siny = Math.sin(Ry);
    var sinz = Math.sin(Rz);

    matrix[0] = cosy*cosz - sinx*siny*sinz;
    matrix[1] = -cosx*sinz;
    matrix[2] = -siny*cosz - sinx*cosy*sinz;

    matrix[4] = cosy*sinz + sinx*siny*cosz;
    matrix[5] = cosx*cosz;
    matrix[6] = -siny*sinz + sinx*cosy*cosz;

    matrix[8] = cosx*siny;
    matrix[9] = -sinx;
    matrix[10] = cosx*cosy;
}

// This method is to reset the matrix to be an identity matrix.
Transform.prototype.reset = function()
{
    this.matrix = [
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1]
}

/*****
/*          Polygon Class          */
*****/
/*
* This class is to create a polygon object.
* @para vertices is an array of Point3D objects. It
* contains a set of Point3D objects arranged in a
* counterclockwise order.
* @para id is an id number for this polygon.
* @para object is the object which this polygon belongs to.
* @para plane is a Plane object which contains the plane
* equation of this polygon.
*/
function Polygon(vertices, id, object, plane) /* Constructor */
{
    this.vertices = vertices;
    this.id = id;

```

```

/* The object this polygon belongs to. */
this.object = object;
this.plane = plane;

// By default, we set an object to be invisible.
this.visibility = invisible;
/*
 * stringOfPoints is containing projected
 * positions in string format which will be later
 * supplied to a "points" attribute of a <polygon>
 * tag.
 */
var len = vertices.length;
if(len == 8)
    this.stringOfPoints = "" + vertices[0].xp + ", " +
        vertices[0].yp + ", " + vertices[1].xp + ", " +
        vertices[1].yp + ", " + vertices[2].xp + ", " +
        vertices[2].yp + ", " + vertices[3].xp + ", " +
        vertices[3].yp + ", " + vertices[4].xp + ", " +
        vertices[4].yp + ", " + vertices[5].xp + ", " +
        vertices[5].yp + ", " + vertices[6].xp + ", " +
        vertices[6].yp + ", " + vertices[7].xp + ", " +
        vertices[7].yp;
else if(len == 4)
    this.stringOfPoints = "" + vertices[0].xp + ", " +
        vertices[0].yp + ", " + vertices[1].xp + ", " +
        vertices[1].yp + ", " + vertices[2].xp + ", " +
        vertices[2].yp + ", " + vertices[3].xp + ", " +
        vertices[3].yp;
else if(len == 3)
    this.stringOfPoints = "" + vertices[0].xp + ", " +
        vertices[0].yp + ", " + vertices[1].xp + ", " +
        vertices[1].yp + ", " + vertices[2].xp + ", " +
        vertices[2].yp;

/* Find the depth of this polygon. */
this.depth = this.getDepth();
}

/*
 * This method is to set a visibility property to this
 * polygon and its polygon vertices.
 * @para value is either 0 or 1 which will be assigned to
 * the visibility property of this object. if value is 0,
 * we set this object to be invisible. If it is 1, we set
 * this object to be visible.
 */
Polygon.prototype.setVisibility = function(value)
{
    this.visibility = value;
    this.plane.visibility = value;

    var v = this.vertices;
    for(var i=0; i<v.length; i++)

```

```

        v[i].visibility = value;
    }

    /*
    * This method is to check if this polygon is a back
    * face. If so, this polygon will appear invisible.
    * @return 1 if this polygon is visible, 0 if this
    * polygon is invisible.
    */
    Polygon.prototype.isVisible = function()
    {
        var plane = this.plane;

        /* Get a plane vector of this polygon. */
        var Na = plane.A;
        var Nb = plane.B;
        var Nc = plane.C;

        /*
        * Get a viewing vector which is create from the
        * viewing reference point and one of polygon
        * vertices.
        */
        Va = this.vertices[0].x3D;
        Vb = this.vertices[0].y3D;
        Vc = this.vertices[0].z3D - g_viewPoint;

        /*
        * The polygon surface will appear visible if the angle
        * between planes and viewing vector is less than 270 but more
        * than 90 degree. So from the dot vector product,  $V1 \cdot V2 = |V1||V2|\cos(\text{angle})$ , we can use the dot product to decide
        * whether of not the polygon is visible or invisible. If the
        * dot product is less than zero, the polygon appears visible.
        * Otherwise, the polygon appears invisible.
        */
        var checkVisible = Va*Na + Vb*Nb + Vc*Nc;

        if(checkVisible < 0)
        { return 1;}
        else
        { return 0; }
    }

    /*
    * This method is to find the approximation depth of this
    * polygon by sampling the middle point on the poloygon
    * surface. The z-value of the middle is used as depth of
    * the polygon.
    * @return a depth of this Polygon object.
    */
    Polygon.prototype.getDepth = function()
    {
        var vertices = this.vertices;

```

```

var len = vertices.length;

if(len == 3)
{
    var depth = (vertices[0].z3D +
                 vertices[1].z3D + vertices[2].z3D)/3;
    return depth;
}
else if(len == 4)
{
    var depth = (vertices[0].z3D + vertices[1].z3D +
                 vertices[2].z3D + vertices[3].z3D)/4;
    return depth;
}
else if(len == 8)
{
    var total = 0;
    for(var i=0; i<8; i++)
        total += vertices[i].z3D;

    var depth = total/8;
    return depth;
}
}

/*****
/*          Plane Class          */
*****/
/*
 * This class is to create a plane equation.
 * @para P1, P2, and P3 are successive Point3D objects
 * arranged in a counterclockwise order.
 */
function Plane(P1, P2, P3)    /* Constructor */
{
    var x1 = P1.x3D;
    var x2 = P2.x3D;
    var x3 = P3.x3D;

    var y1 = P1.y3D;
    var y2 = P2.y3D;
    var y3 = P3.y3D;

    var z1 = P1.z3D;
    var z2 = P2.z3D;
    var z3 = P3.z3D;

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

    this.visibility = invisible;

```

```

    }
    /*****
    /*          Vector Class          */
    /*****/
    /*
    * This class is to create a vector object.
    * @para A, B, and C are Cartesian components from the
    * plane equation  $Ax+By+Cz+D = 0$ .
    */
    function Vector(A, B, C) /* Constructor */
    {
        this.A = A;
        this.B = B;
        this.C = C;
    }

    /*
    * This method is to return a size of a Vector object.
    * @return a size of a Vector object.
    */
    Vector.prototype.size = function()
    {
        return Math.sqrt((this.A*this.A) +
            (this.B*this.B) + (this.C*this.C));
    }

    /*
    * This method is to normalize a Vector object to be a
    * unit vector.
    */
    Vector.prototype.normalize = function()
    {
        var size = Math.sqrt((this.A*this.A) +
            (this.B*this.B) + (this.C*this.C));
        this.A /= size;
        this.B /= size;
        this.C /= size;
    }

    /*
    * This method is to add another Vector object to this
    * Vector object.
    * @para a_vector is another Vector object that we want
    * to add.
    */
    Vector.prototype.addVector = function(a_vector)
    {
        this.A += a_vector.A;
        this.B += a_vector.B;
        this.C += a_vector.C;
    }

    /*
    * This method is to find the dot product of two Vector

```

```

* objects.
* @para a_vector is another Vector object that we want
* to do dot product with.
* @return a scalar value of the dot product of these two
* vector objects.
*/
Vector.prototype.dotVector = function(a_vector)
{
    return this.A*a_vector.A + this.B*a_vector.B +
           this.C*a_vector.C;
}
/*****
/*          Primitive Class.          */
/* This class is a superclass of Box, Cone, Cylinder, and Sphere.      */
/*****/
/*
* The Primitive class is a super class of a box, a cone,
* a cylinder, and a sphere object. This class contains common
* properties and methods shared among four subclasses.
* @para thisNode is either <Box>, <Cone>, <Cylinder>, or
* <Sphere> node from a parse structure of a source document.
* @para appearanceNode is an <appearance> node where the
* color of the object is specified.
* @para id is an id for this primitive.
*/
function Primitive(thisNode, appearanceNode, id)
{
    if( arguments.length > 0)
        this.init(thisNode, appearanceNode, id);
}

/*
* This method is to initialize properties to this
* object.
* @para thisNode is either <Box>, <Cone>, <Cylinder>, or
* <Sphere> node from a parse structure of a source document.
* @para appearanceNode is an <appearance> node where the
* color of the object is specified.
* @para id is an id for this primitive.
*/
Primitive.prototype.init = function(thisNode,
    appearanceNode, id)
{
    this.node = thisNode;
    this.appearanceNode = appearanceNode;
    this.id = id;

    var shape = thisNode.nodeName;

    this.parent = thisNode.parentNode;

    /*
    * A "center1" property is the center of the box
    * before users change perspective.

```



```

*/
this.center1 = new Point3D(0, 0, 0);

/*
 * A "center2" property is the current center of
 * the box after users change perspective.
 */
this.center2 = new Point3D(0, 0, 0);

this.num_vertices = 0;
this.num_faces = 0;

/*
 * A number of vertices and polygons to form the
 * primitive are different and depend on a type of
 * primitive object.
 */
if(shape == "Box")
{
    this.num_vertices = 8;
    this.num_faces = 6;
}
else if(shape == "Cone")
{
    this.num_vertices = 33;
    this.num_faces = 33;
}
else if(shape == "Cylinder")
{
    this.num_vertices = 16;
    this.num_faces = 10;
}
else if(shape == "Sphere")
{
    this.num_vertices = 58;
    this.num_faces = 64;
}

/* Get color of this object. */
if(appearanceNode != null)
{
    var tmp =appearanceNode.getElementsByTagName("Material");
    var material = tmp[0];

    var tmp1 = material.getAttribute("diffuseColor");
    var color = tmp1.split(' ');
    this.rgb = new Array(Number(color[0]), Number(color[1]),
        Number(color[2]));
}
else
    this.rgb = new Array(0.3, 0.6, 0.9);    // default color

/*
 * An "wc" array contains world coordinate of

```

```

    * polygon vertices.
    */
    this.wc = new Array(this.num_vertices);
    for(var i=0; i<this.num_vertices; i++)
        this.wc[i] = new Point3D(0, 0, 0);

    /*
    * A "vc" array contains viewing coordinates of
    * polygon vertices.
    */
    this.vc = new Array(this.num_vertices);
    for(var i=0; i<this.num_vertices; i++)
        this.vc[i] = new Point3D(0, 0, 0);

    /*
    * A "projection" array contains projection
    * coordinates of polygon vertices.
    */
    this.projection = new Array(this.num_vertices);
    for(var i=0; i<this.num_vertices; i++)
        this.projection[i] = new Point3D(0,0,0);

    /*
    * A "face" array contains a set of polygon
    * vertices arranged in a counterclockwise order
    * and associated with that face.
    */
    this.face = new Array(this.num_faces)
    for(var i=0; i<this.num_faces; i++)
        this.face[i] = new Array(4);

    /*
    * A "plane" array contains a plane equation of
    * each polygon.
    */
    this.plane = new Array(this.num_faces);

    /*
    * An "avgNormalVectors" array contains an average
    * unit normal vector of each vertex.
    */
    this.avgNormalVectors = new Array(this.num_vertices);

    /*
    * An "I" array contains a light intensity of each
    * vertex.
    */
    this.intensity = new Array(this.num_vertices);

    /*
    * A "color_left" and "color_right" contains the
    * RGB color of a left and right edge of each
    * polygon.
    */

```

```

this.color_left = new Array(this.num_faces);
this.color_right = new Array(this.num_faces);

/*
 * Create SVG <polygon> and <linearGradient> tags
 * and inserted these tags to the root SVG node.
 */
this.createTag();
}

/*
 * This method is to create SVG <polygon> and
 * <linearGradient> tags. Once all tags are created, they
 * will be inserted into the <svg> root node.
 */
Primitive.prototype.createTag = function()
{
    /*
     * Create a "defs" element where <linearGradient>
     * tags will be inserted as children nodes.
     */
    var defs = document.createElement("svg:defs");
    // Append a <defs> tag into the <svg> root node.
    g_svg.appendChild(defs);

    var polygons = new Array(this.num_faces);
    var points = new Array(this.num_faces);
    var fill = new Array(this.num_faces);
    var stroke = new Array(this.num_faces);

    var linearGradient = new Array(this.num_faces);
    var id = new Array(this.num_faces);
    var stopL = new Array(this.num_faces);
    var stopR = new Array(this.num_faces);
    var offsetL = new Array(this.num_faces);
    var offsetR = new Array(this.num_faces);
    var stop_colorL = new Array(this.num_faces);
    var stop_colorR = new Array(this.num_faces);

    /*
     * Create <polygon> tags for building a wireframe
     * representation of objects and <linearGradient>
     * tags for shading a polygon surface.
     */
    for(var k=0; k<this.num_faces; k++)
    {
        /* Create a "polygon" element. */
        polygons[k] = document.createElement("svg:polygon");

        /*
         * Create attributes of the "polygon" element and assign
         * values to each attribute in such a way that this
         * polygon appears invisible.
         */
    }

```

```

points[k] = document.createAttribute("points");
points[k].value = "0, 0";
fill[k] = document.createAttribute("fill");
fill[k].value = "none";
stroke[k] = document.createAttribute("stroke");
stroke[k].value = "none";

/*
 * Bind polygon's attribute nodes to the
 * "polygon" element
 */
polygons[k].attributes.setNamedItem(points[k]);
polygons[k].attributes.setNamedItem(fill[k]);
polygons[k].attributes.setNamedItem(stroke[k]);

/*
 * Store a <polygon> tag in a global
 * variable, which will be used later to
 * draw this polygon.
 */
g_polygonSVGTag[g_countTag++] = polygons[k];

/*
 * Append a <polygon> tag into the <svg>
 * root node.
 */
g_svg.appendChild(polygons[k]);

/*
 * Create a "linearGradient" element for
 * shading.
 */
linearGradient[k] =
    document.createElement("svg:linearGradient");

/*
 * Create an id attribute of a "linearGradient" element.
 * The id of this tag is used later when a <polygon> tag
 * refers to this <linearGradient> tag. we use an object
 * Id and a polygon Id to identify a unique id for the
 * <linearGradient> tag.
 */
id[k] = document.createAttribute("id");
id[k].value = "" + this.id + k;

//Bind an "id" attribute to the "linearGradient" element.
linearGradient[k].attributes.setNamedItem(id[k]);

// Append a <linearGradient> tag to a <defs> tag.
defs.appendChild(linearGradient[k]);

/*
 * Create a "stop" element and an "offset" attribute where
 * we can define starting and stopping colors on the left

```

```

    * and right edge of each polygon surface for shading.
    */
    stopL[k] = document.createElement("svg:stop");
    offsetL[k] = document.createAttribute("offset");
    offsetL[k].value = "0%";
    stop_colorL[k] = document.createAttribute("stop-color");

    // Bind stop's attribute nodes to the "stop" element
    stopL[k].attributes.setNamedItem(offsetL[k]);
    stopL[k].attributes.setNamedItem(stop_colorL[k]);
    stopR[k] = document.createElement("svg:stop");
    offsetR[k] = document.createAttribute("offset");
    offsetR[k].value = "100%";
    stop_colorR[k] = document.createAttribute("stop-color");
    stopR[k].attributes.setNamedItem(offsetR[k]);
    stopR[k].attributes.setNamedItem(stop_colorR[k]);

    /* Append a <stop> tag to a <linearGradient> tag. */
    linearGradient[k].appendChild(stopL[k]);
    linearGradient[k].appendChild(stopR[k]);
}

}

/*
 * This method is to modify world coordinates of polygon vertices
 * according to <transform> and <group> parent nodes.
 */
Primitive.prototype.modifyWC = function()
{
    var wc = this.wc;
    /*
    * Check if there is any transform parent tags because it will
    * affect the world coordinates and the center of this object.
    */
    var contextNode = this.parent;
    var parentNode = contextNode.parentNode;

    while(parentNode.nodeName != "Scene")
    {
        if(parentNode.nodeName == "Transform")
        {
            // Get translation, rotation, and scale values.
            var translation =
                parentNode.getAttribute("translation");
            var rotation =
                parentNode.getAttribute("rotation");
            var scale =
                parentNode.getAttribute("scale");

            /* Get translation distances.*/
            var temp = translation.split(' ');
            var Tx = Number(temp[0]);
            var Ty = Number(temp[1]);
            var Tz = Number(temp[2]);

```

```

        this.center1.x3D += Tx;
        this.center1.y3D += Ty;
        this.center1.z3D += Tz;

        this.center2.x3D = this.center1.x3D;
        this.center2.y3D = this.center1.y3D;
        this.center2.z3D = this.center1.z3D;

        /* Get rotation angles. */
        temp = rotation.split(' ');
        var Rx = Number(temp[0])*Number(temp[3]);
        var Ry = Number(temp[1])*Number(temp[3]);
        var Rz = Number(temp[2])*Number(temp[3]);

        /* Get scaling factors. */
        temp = scale.split(' ');
        var Sx = Number(temp[0]);
        var Sy = Number(temp[1]);
        var Sz = Number(temp[2]);

        /* Get a T*R transformation matrix. */
        var transform = new Transform();
        var matrix = transform.matrix;
        transform.translate(Tx, Ty, Tz);
        transform.rotate(Rx, Ry, Rz);

        /* Get new transformed WC = R*T*S*WC. */
        for(var i=0; i<this.num_vertices; i++)
        {
            /* Get S*WC. */
            wc[i].scale(Sx, Sy, Sz)
            wc[i].translate(matrix);
            wc[i].rotate(matrix);
        }
        contextNode = parentNode;
        parentNode = contextNode.parentNode;
    }
}
/*
 * This method is to convert world coordinates to viewing
 * coordinates.
 */
Primitive.prototype.createVC = function()
{
    var matrix = perspective.matrix;

    var wc = this.wc;
    var vc = this.vc;
    for(var i=0; i<this.num_vertices; i++)
    {
        vc[i].setXYZ(wc[i].x3D, wc[i].y3D, wc[i].z3D);
        vc[i].rotate_translate(matrix);
    }
}

```

```

    }
}

/*
 * This method is to convert view coordinates to projection
 * coordinates.
 */
Primitive.prototype.createProjection = function()
{
    var vc = this.vc;
    var projection = this.projection;

    for(var i=0; i<this.num_vertices; i++)
    {
        projection[i].setXYZ(vc[i].x3D, vc[i].y3D, vc[i].z3D);
        projection[i].project(g_viewPoint);
    }
}

/*
 * This method is to adjust the center of the object when
 * perspectives has been changed.
 */
Primitive.prototype.adjustCenter = function()
{
    var center1 = new Point3D(this.center1.x3D, this.center1.y3D,
        this.center1.z3D);
    var matrix = perspective.matrix;

    center1.translate(matrix);
    center1.rotate(matrix);

    /*
     * Update this.current2 which is to keep new coordinates of
     * the center of an object when changing perspectives occur.
     */
    this.center2.x3D = center1.x3D;
    this.center2.y3D = center1.y3D;
    this.center2.z3D = center1.z3D;
}

/*
 * This method is to calculate the plane equation for each
 * polygon surface.
 */
Primitive.prototype.createPlane = function()
{
    for(var i=0; i<this.num_faces; i++)
        this.plane[i] = new Plane(this.face[i][0],
            this.face[i][1], this.face[i][2]);
}

// This method is to find the intensity of each polygon vertex.
Primitive.prototype.findIntensity = function()

```

```

{
    /*
    * We used this formula to find the intensity of each vertex.
    *  $I = I_a + I_d + I_s$  or  $I = p_a + p_d(l \cdot n) + p_s(h \cdot n)^2$ 
    * where l is a light vector, n is an average normal plane
    * vector of the vertex,  $p_a + p_d + p_s = 1$  ,
    *  $h = (l + v) / |l + v|$ , and v is a viewing vector.
    *
    * In this case, l and v are the same vectors because we
    * assume that a point light source is coming from the viewing
    * point.
    */
    var centerPoint = this.center2;

    var l = new Vector(centerPoint.x3D, centerPoint.y3D,
        centerPoint.z3D - g_viewPoint);
    l.normalize();
    var h = new Vector(l.A, l.B, l.C);
    h.addVector(l);
    h.normalize();
    var n = this.avgNormalVectors;
    var I = this.intensity;
    var pa = 0.2;
    var pd = 0.6;
    var ps = 0.2;
    var p = this.projection;
    /* Calculate the intensity of each vertex. */
    for(var i=0; i<this.num_vertices; i++)
    {
        /*
        * Calculate only light intensity of a visible polygon
        * vertex.
        */
        if(p[i].visibility == visible)
        {
            var Id = new Vector(l.A, l.B, l.C);
            l_dot_n = Id.dotVector(n[i]);

            var Is = new Vector(h.A, h.B, h.C);
            h_dot_n = Is.dotVector(n[i]);

            I[i] = pa + pd*l_dot_n + ps*(h_dot_n * h_dot_n);
        }
    }
}

/*
* This method is to supply colors in sRGB format to existing
* <linearGradient> tags as stop colors on the left and right
* edge of a polygon surfaces.
*/
Primitive.prototype.modifyGradientTag = function()
{
    var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

```



```

var p = this.plane;

for(var i=0; i<this.num_faces; i++)
{
    if(p[i].visibility == visible)
    {
        var shadeId = "" + this.id + i;
        var shadeTag = document.getElementById(shadeId);
        var leftStop = shadeTag.firstChild;
        var rightStop = shadeTag.lastChild;

        /* Get the rgb color on each edge of a polygon. */
        var rgb_left = hsvTorgb(hsv[0], hsv[1],
            this.color_left[i]);
        var rgb_right = hsvTorgb(hsv[0], hsv[1],
            this.color_right[i]);

        /*
        * Change the format to be sRGB which is legally
        * used in SVG.
        */
        rgb_left[0] *= 255;
        rgb_left[1] *= 255;
        rgb_left[2] *= 255;
        rgb_right[0] *= 255;
        rgb_right[1] *= 255;
        rgb_right[2] *= 255;

        var c_l = "rgb(" + rgb_left[0] + ", " + rgb_left[1] +
            ", " + rgb_left[2] + ")";
        var c_r = "rgb(" + rgb_right[0] + ", " +
            rgb_right[1] + ", " + rgb_right[2] + ")";

        /* Supply stop colors to a <linearGradient> tag. */
        leftStop.setAttribute("stop-color", c_l);
        rightStop.setAttribute("stop-color", c_r);
    }
}

}

/*****
/*      Box Class, a subclass of Primitive class      */
/*****
/*
* When we access an object property, the interpreter will look
* at the current object's properties to see if one by that name
* exists. If the name does not exist there, then the interpreter
* looks at the prototype property of the object to see if that
* object, the one pointed to by the prototype property, has the
* named property. If there is no property there, then the
* interpreter looks to see if the prototype property has a
* prototype property. If it does, then this process continues
* until either the property is found or until there are no more
* prototype properties to search. We use a prototype property to
* make OOP in JavaScript. We first set up the prototype

```

```

* inheritance chain. Whenever we lookup this property
* in the object, if it cannot be found, then the interpreter
* will work its way up the inheritance chain to look for that
* property. Primitive will be the first object in the chain.
*/
Box.prototype = new Primitive();

/*
* The constructor property is to determine an object's type.
* After we redefined a prototype property for a Box object, we
* also changed the constructor to Primitive. We need to redefine
* the constructor property to be Box.
*/
Box.prototype.constructor = Box;

/*
* This code allows us to call methods in a superclass that are
* hidden by redefined methods in subclass.
*/
Box.superclass = Primitive.prototype;

/*
* This class is to create a Box object. Three parameters are
* also given to a superclass Primitive.
* @para thisNode is either <Box>, <Cone>, <Cylinder>, or
* <Sphere> node from a parse structure of a source document.
* This property will be set by a super-class init method.
* @para appearanceNode is an <appearance> node where the color
* of the object is specified. And this property is set by a
* super-class init method.
* @para id is a unique number assigned as id for a Box object.
*/
function Box(thisNode, appearanceNode, id) /* Constructor */
{
    if(arguments.length > 0)
        this.init(thisNode, appearanceNode, id);
}

/*
* This method is to initialize values to the object's properties
* @para thisNode is either <Box>, <Cone>, <Cylinder>, or
* <Sphere> node from a parse structure of a source document.
* @para appearanceNode is an <appearance> node where the color
* of the object is specified.
* @para id is a unique number assigned as id for a Box object.
*/
Box.prototype.init = function(thisNode, appearanceNode, id)
{
    /*
    * Call a super-class init method to set thisNode,
    * appearanceNode, and id for this object.
    */
    Box.superclass.init.call(this, thisNode, appearanceNode, id);
}

```

```

var size = thisNode.getAttribute("size");
var dimension = size.split(' ');
this.x = Number(dimension[0]);
this.y = Number(dimension[1]);
this.z = Number(dimension[2]);

/* avgI contains the avg intensity of each box's edge. */
this.avgI = new Array(this.num_vertices);

this.createFirstWC();
this.modifyWC();
}

/*
 * This method is to model word-coordinate vertices before doing
 * any transformation.
 */
Box.prototype.createFirstWC = function()
{
    var wc = this.wc;
    var x = this.x;
    var y = this.y;
    var z = this.z;

    /*
     * The box is created center at the origin.
     * so the x, y, and z coordinates of each corner are a half of
     * the x, y, and z dimension. Index 0 - 3 indicate the corner
     * of the front side starting from the upper-left corner in a
     * clockwise direction. Index 4 - 7 indicate the corner of the
     * back side starting from the upper-left corner in a
     * clockwise direction.
     */
    wc[0].setXYZ(-x/2, y/2, z/2);
    wc[1].setXYZ(x/2, y/2, z/2);
    wc[2].setXYZ(x/2, -y/2, z/2);
    wc[3].setXYZ(-x/2, -y/2, z/2);
    wc[4].setXYZ(-x/2, y/2, -z/2);
    wc[5].setXYZ(x/2, y/2, -z/2);
    wc[6].setXYZ(x/2, -y/2, -z/2);
    wc[7].setXYZ(-x/2, -y/2, -z/2);
}

/* This method is to model polygon vertices on each surface. */
Box.prototype.assignProjectionsToFaces = function()
{
    var projection = this.projection;

    /* front face */
    this.face[0] = [projection[0], projection[3], projection[2],
        projection[1]];
    /* back face */
    this.face[1] = [projection[5], projection[6], projection[7],
        projection[4]];
}

```

```

    /* left face */
    this.face[2] = [projection[4], projection[7], projection[3],
        projection[0]];
    /* right face */
    this.face[3] = [projection[1], projection[2], projection[6],
        projection[5]];
    /* top face */
    this.face[4] = [projection[4], projection[0], projection[1],
        projection[5]];
    /* bottom face */
    this.face[5] = [projection[3], projection[7], projection[6],
        projection[2]];
}

/*
 * This method is to calculate the average normal vector of each
 * vertex. Only a visible vertex is calculated.
 */
Box.prototype.createNormalVector = function()
{
    var p = this.plane;
    var n = this.avgNormalVectors;

    var front = 0;
    var back = 1;
    var left = 2;
    var right = 3;
    var top = 4;
    var bottom = 5;

    var f = new Vector(p[front].A, p[front].B, p[front].C);
    var b = new Vector(p[back].A, p[back].B, p[back].C);
    var l = new Vector(p[left].A, p[left].B, p[left].C);
    var r = new Vector(p[right].A, p[right].B, p[right].C);
    var t = new Vector(p[top].A, p[top].B, p[top].C);
    var butt = new Vector(p[bottom].A, p[bottom].B, p[bottom].C);

    var v = this.projection;

    /*
     * Create an average normal vector of vertex 0 which is an
     * average combination of a front, top, and left plane.
     */
    if(v[0].visibility == visible)
    {
        n[0] = new Vector(p[front].A, p[front].B, p[front].C);
        n[0].addVector(t);
        n[0].addVector(l);
        n[0].normalize();
    }

    /*
     * Create an average normal vector of vertex 1 which is an
     * average combination of a front, top, and right plane. */

```

```

if(v[1].visibility == visible)
{
    n[1] = new Vector(p[front].A, p[front].B, p[front].C);
    n[1].addVector(t);
    n[1].addVector(r);
    n[1].normalize();
}

/*
 * Create an average normal vector of vertex 2 which is an
 * average combination of a front, back, and right plane.
 */
if(v[2].visibility == visible)
{
    n[2] = new Vector(p[front].A, p[front].B, p[front].C);
    n[2].addVector(b);
    n[2].addVector(r);
    n[2].normalize();
}

/*
 * Create an average normal vector of vertex 3 which is an
 * average combination of a front, back, and left plane.
 */
if(v[3].visibility == visible)
{
    n[3] = new Vector(p[front].A, p[front].B, p[front].C);
    n[3].addVector(b);
    n[3].addVector(l);
    n[3].normalize();
}

/*
 * Create an average normal vector of vertex 4 which is an
 * average combination of a back, top, and left plane.
 */
if(v[4].visibility == visible)
{
    n[4] = new Vector(p[back].A, p[back].B, p[back].C);
    n[4].addVector(t);
    n[4].addVector(l);
    n[4].normalize();
}

/*
 * Create an average normal vector of vertex 5 which is an
 * average combination of a back, top, and right plane.
 */
if(v[5].visibility == visible)
{
    n[5] = new Vector(p[back].A, p[back].B, p[back].C);
    n[5].addVector(t);
    n[5].addVector(r);
}

```

```

        n[5].normalize();
    }

    /*
    * Create an average normal vector of vertex 6 which is an
    * average combination of a back, bottom, and right plane.
    */
    if(v[6].visibility == visible)
    {
        n[6] = new Vector(p[back].A, p[back].B, p[back].C);
        n[6].addVector(butt);
        n[6].addVector(r);
        n[6].normalize();
    }

    /*
    * Create an average normal vector of vertex 7 which is an
    * average combination of a back, bottom, and left plane.
    */
    if(v[7].visibility == visible)
    {
        n[7] = new Vector(p[back].A, p[back].B, p[back].C);
        n[7].addVector(butt);
        n[7].addVector(l);
        n[7].normalize();
    }
}

/*
* This method is to calculate RGB colors that are used as
* starting and stopping colors in <linearGradient> tags which
* performs a linear interpolation on polygon surfaces.
*/
Box.prototype.findColor = function()
{
    var avgI = this.avgI;
    var I = this.intensity;

    var p = this.projection;

    /*
    * Calculate the average intensity of each edge. Only a
    * visible vertex is calculated.
    */
    if(p[0].visibility == visible)
        avgI[0] = (I[0] + I[3])/2;
    if(p[1].visibility == visible)
        avgI[1] = (I[1] + I[2])/2;
    if(p[2].visibility == visible)
        avgI[2] = (I[5] + I[6])/2;
    if(p[3].visibility == visible)
        avgI[3] = (I[4] + I[7])/2;
    if(p[4].visibility == visible)
        avgI[4] = (I[4] + I[0])/2;

```

```

    if(p[5].visibility == visible)
        avgI[5] = (I[5] + I[1])/2;
    if(p[6].visibility == visible)
        avgI[6] = (I[3] + I[7])/2;
    if(p[7].visibility == visible)
        avgI[7] = (I[2] + I[6])/2;

    /* Get the RGB color and translate into HSV format. */
    var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

    var front = 0; var back = 1; var left = 2; var right = 3;
    var top = 4; var bottom = 5;

    var color_left = this.color_left;
    var color_right = this.color_right;

    var plane = this.plane;

    /* Colors on each edge are in HSV format. */
    if(plane[front].visibility == visible)
    {
        color_left[front] = hsv[2] - avgI[0]*(0.5);
        color_right[front] = hsv[2] - avgI[1]*(0.5);
    }
    if(plane[back].visibility == visible)
    {
        color_left[back] = hsv[2] - avgI[2]*(0.5);
        color_right[back] = hsv[2] - avgI[3]*(0.5);
    }
    if(plane[left].visibility == visible)
    {
        color_left[left] = hsv[2] - avgI[3]*(0.5);
        color_right[left] = hsv[2] - avgI[0]*(0.5);
    }
    if(plane[right].visibility == visible)
    {
        color_left[right] = hsv[2] - avgI[1]*(0.5);
        color_right[right] = hsv[2] - avgI[2]*(0.5);
    }
    if(plane[top].visibility == visible)
    {
        color_left[top] = hsv[2] - avgI[4]*(0.5);
        color_right[top] = hsv[2] - avgI[5]*(0.5);
    }
    if(plane[bottom].visibility == visible)
    {
        color_left[bottom] = hsv[2] - avgI[6]*(0.5);
        color_right[bottom] = hsv[2] - avgI[7]*(0.5);
    }
}

/*
 * This method is to gather other methods for building a Box
 * object. Whenever perspectives have been changed, only this

```

```

* modify method is called to re-compute viewing coordinates,
* projection, and color intensities and re-assign new values to
* each face.
*/
Box.prototype.modify = function()
{
    this.createVC();           /* super class's method */
    this.createProjection();    /* super class's method */
    this.assignProjectionsToFaces();
    this.adjustCenter()        /* super class's method */
    this.createPlane();         /* super class's method */
    this.createNormalVector();
    this.findIntensity();       /* super class's method */
    this.findColor();
    this.modifyGradientTag();   /* super class's method */
}

Box.prototype.getVCandProjection = function()
{
    this.createVC();           /* super class's method */
    this.createProjection();    /* super class's method */
    this.assignProjectionsToFaces();
    this.adjustCenter()        /* super class's method */
    this.createPlane();         /* super class's method */
}

Box.prototype.getShade = function()
{
    this.createNormalVector();
    this.findIntensity(); /* super class's method */
    this.findColor();
    this.modifyGradientTag(); /* super class's method */
}

/*****
/*      Cone Class, a subclass of Primitive
* All methods of this class are doing the same things as Box's methods
* do but purposely for a Cone object . So the way we model world
* coordinates and assign them to each surface is different from the
* other kinds of objects. However, the concept of calculating viewing
* coordinates and color intensities remains the same.
*/
*****/
Cone.prototype = new Primitive();
Cone.prototype.constructor = Cone;
Cone.superclass = Primitive.prototype;

function Cone(thisNode, appearanceNode, id)
{
    if(arguments.length > 0)
        this.init(thisNode, appearanceNode, id);
}

Cone.prototype.init = function(thisNode, appearanceNode, id)

```



```

{
    /* Call a superclass init. */
    Cone.superclass.init.call(this, thisNode, appearanceNode,
        id);

    this.bottomRadius = thisNode.getAttribute("bottomRadius");
    this.height = thisNode.getAttribute("height");

    /* avgI contains the avg intensity of each polygon's edge. */
    this.avgI = new Array(this.num_vertices - 1);

    this.createFirstWC();
    this.modifyWC();
}

Cone.prototype.createFirstWC = function()
{
    var r = this.bottomRadius;
    var h = this.height;
    var wc = this.wc;
    var index = 0;

    var cos = new Array(8);      /* Store the cos data. */
    cos[0] = 1;
    cos[1] = Math.cos(Math.PI/4);
    cos[2] = 0;
    cos[3] = -cos[1];
    cos[4] = -1;
    cos[5] = -cos[1];
    cos[6] = 0;
    cos[7] = cos[1];

    var sin = new Array(8);      /* Store the sin data. */
    sin[0] = 0;
    sin[1] = -cos[1];           /* cos(pi/4) == sin(pi/4) */
    sin[2] = -1;
    sin[3] = sin [1];
    sin[4] = 0;
    sin[5] = -sin[1];
    sin[6] = 1;
    sin[7] = -sin[1];

    var height_step = 0;
    var heightFactor = 4;
    var radiusFactor = 4;
    var angleFactor = 8;

    for(var i=radiusFactor; i>0; i--)
    {
        for(var k=0; k<angleFactor; k++)
        {
            var x = (i*r/radiusFactor)*cos[k];
            var y = height_step*h/heightFactor;
            var z = (i*r/radiusFactor)*sin[k];

```

```

        wc[index++].setXYZ(x, y, z);
    }
    height_step++;
}

/* the highest point */
wc[index].setXYZ(0, h, 0);
}

Cone.prototype.assignProjectionsToFaces = function()
{
    var face = this.face;
    var p = this.projection;

    var num_faces = this.num_faces;

    /* Side rectangular polygons. */
    for(var i=0; i<num_faces - 8; i++)
    {
        if( ((i + 1) % 8) == 0)
            face[i] = [p[i], p[i-7], p[i+1], p[i+8]];
        else
            face[i] = [p[i], p[i+1], p[i+9], p[i+8]];
    }

    /* Triangles are at the top */
    for(var i=24; i<num_faces - 1; i++)
    {
        if(i == num_faces - 2)
            face[i] = [p[i], p[24], p[32]];
        else
            face[i] = [p[i], p[i+1], p[32]];
    }

    /* The bottom part of a cone. */
    face[32] = [p[7], p[6], p[5], p[4], p[3], p[2], p[1], p[0]];
}

Cone.prototype.createNormalVector = function()
{
    var p = this.plane;
    var n = this.avgNormalVectors;

    var v = this.projection;

    /*
    * An average normal vector at vertex 0 is an average vector
    * of plane 0 and 7.
    */
    if(v[0].visibility == visible)
    {
        var adj1 = new Vector(p[7].A, p[7].B, p[7].C);
        n[0] = new Vector(p[0].A, p[0].B, p[0].C);
    }
}

```

```

        n[0].addVector(adj1);
        n[0].normalize();
    }

    /* Find an average normal vector of vertex 1 to 7. */
    for(var i=1; i<=7; i++)
    {
        if(v[i].visibility == visible)
        {
            var adj1 = new Vector(p[i-1].A, p[i-1].B,
                                  p[i-1].C);
            n[i] = new Vector(p[i].A, p[i].B, p[i].C);
            n[i].addVector(adj1);
            n[i].normalize();
        }
    }

    /* Find an average normal vector of vertex 8 to 31. */
    for(var i=8; i<=31; i++)
    {
        if(v[i].visibility == visible)
        {
            var adj1 = new Vector(p[i-8].A, p[i-8].B,
                                  p[i-8].C);
            var adj2 = new Vector(p[i-1].A, p[i-1].B,
                                  p[i-1].C);
            var adj3 = "";

            if( i%8 == 0)
                adj3 = new Vector(p[i+7].A, p[i+7].B,
                                  p[i+7].C);
            else
                adj3 = new Vector(p[i-9].A, p[i-9].B,
                                  p[i-9].C)

            n[i] = new Vector(p[i].A, p[i].B, p[i].C);
            n[i].addVector(adj1);
            n[i].addVector(adj2);
            n[i].addVector(adj3);
            n[i].normalize();
        }
    }

    var num_vertices = this.num_vertices;
    /* Find an average normal vector of vertex 32. */
    if(v[32].visibility == visible)
    {
        var adj1 = new Vector(p[24].A, p[24].B, p[24].C);
        var adj2 = new Vector(p[25].A, p[25].B, p[25].C);
        var adj3 = new Vector(p[26].A, p[26].B, p[26].C);
        var adj4 = new Vector(p[27].A, p[27].B, p[27].C);
        var adj5 = new Vector(p[28].A, p[28].B, p[28].C);
        var adj6 = new Vector(p[29].A, p[29].B, p[29].C);
        var adj7 = new Vector(p[30].A, p[30].B, p[30].C);
    }

```

```

        n[32] = new Vector(p[31].A, p[31].B, p[31].C);
        n[32].addVector(adj1);
        n[32].addVector(adj2);
        n[32].addVector(adj3);
        n[32].addVector(adj4);
        n[32].addVector(adj5);
        n[32].addVector(adj6);
        n[32].addVector(adj7);
        n[32].normalize();
    }
}

Cone.prototype.findColor = function()
{
    var avgI = this.avgI;
    var I = this.intensity;

    var p = this.projection;
    var num_faces = this.num_faces;

    /* Calculate the average intensity of each edge. */
    for(var i=0; i<=23; i++)
        if(p[i].visibility == visible)
            avgI[i] = (I[i] + I[i+8])/2;

    for(var i=24; i<=31; i++)
        if(p[i].visibility == visible)
            avgI[i] = (I[i] + I[32])/2;

    /*
    * Get the RGB color of the Box and translate into HSV format.
    */
    var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

    var color_left = this.color_left;
    var color_right = this.color_right;

    var plane = this.plane;

    /* Get a color on each edge in hsv format. */
    for(var i=0; i<=31; i++)
    {
        if(plane[i].visibility == visible)
        {
            if(i%8 == 0)
                color_left[i] = hsv[2] - avgI[i]*(0.5);
            else if(plane[i-1].visibility == visible)
                color_left[i] = color_right[i-1];
            else
                color_left[i] = hsv[2] - avgI[i]*(0.5);

            if((i+1)%8 == 0 && plane[i-7].visibility == visible)
                color_right[i] = color_left[i-7];
            else if((i+1)%8 == 0 &&

```

```

        plane[i-7].visibility == invisible)
        color_right[i] = hsv[2] - avgI[i-7]*(0.5);
    else
        color_right[i] = hsv[2] - avgI[i+1]*(0.5);
    }
}

/* Get a color for the bottom of the cone. */
if(plane[32].visibility == visible)
{
    color_left[32] = hsv[2] - I[4]*(0.5);
    color_right[32] = hsv[2] - I[0]*(0.5);
}
}

Cone.prototype.modify = function()
{
    this.createVC();           /* super class's method */
    this.createProjection();    /* super class's method */
    this.assignProjectionsToFaces();
    this.adjustCenter()        /* super class's method */
    this.createPlane();         /* super class's method */
    this.createNormalVector();
    this.findIntensity();        /* super class's method */
    this.findColor();
    this.modifyGradientTag();    /* super class's method */
}

Cone.prototype.getVCandProjection = function()
{
    this.createVC();           /* super class's method */
    this.createProjection();    /* super class's method */
    this.assignProjectionsToFaces();
    this.adjustCenter()        /* super class's method */
    this.createPlane();         /* super class's method */
}

Cone.prototype.getShade = function()
{
    this.createNormalVector();
    this.findIntensity();        /* super class's method */
    this.findColor();
    this.modifyGradientTag();    /* super class's method */
}

/*****
/*
/*      Sphere Class, a subclass of Primitive
/* All methods of this class are doing the same things as Box's methods
/* do but purposely for a Sphere object. So the way we model world
/* coordinates and assign them to each surface is different from the
/* other kinds of objects. However, the concept of calculating viewing
/* coordinates and color intensities remains the same.
*/
*****/
Sphere.prototype = new Primitive();

```

```

Sphere.prototype.constructor = Sphere;
Sphere.superclass = Primitive.prototype;

function Sphere(thisNode, appearanceNode, id)
{
    if(arguments.length > 0)
        this.init(thisNode, appearanceNode, id);
}

Sphere.prototype.init = function(thisNode, appearanceNode, id)
{
    /* Call a superclass init. */
    Sphere.superclass.init.call(this, thisNode, appearanceNode,
        id);

    this.radius = thisNode.getAttribute("radius");

    /* avgI contains the avg intensity of each polygon's edge. */
    this.avgI = new Array(this.num_vertices - 2);

    this.createFirstWC();
    this.modifyWC();
}

Sphere.prototype.createFirstWC = function()
{
    var wc = this.wc;
    var r = this.radius;

    var cos = new Array(8); /* Store the cos data. */
    cos[0] = 1;
    cos[1] = Math.cos(Math.PI/4);
    cos[2] = 0;
    cos[3] = -cos[1];
    cos[4] = -1;
    cos[5] = -cos[1];
    cos[6] = 0;
    cos[7] = cos[1];

    var sin = new Array(8); /* Store the sin data. */
    sin[0] = 0;
    sin[1] = -cos[1]; /* cos(pi/4) == sin(pi/4) */
    sin[2] = -1;
    sin[3] = sin [1];
    sin[4] = 0;
    sin[5] = -sin[1];
    sin[6] = 1;
    sin[7] = -sin[1];

    var h_step = new Array(9); /* Height steps. */
    h_step[0] = r;
    h_step[1] = r*Math.sin(3*Math.PI/8);
    h_step[2] = r*Math.sin(Math.PI/4);
    h_step[3] = r*Math.sin(Math.PI/8);

```

```

h_step[4] = 0;
h_step[5] = -h_step[3];
h_step[6] = -h_step[2];
h_step[7] = -h_step[1];
h_step[8] = -r;

var r_step = new Array(9);
r_step[0] = 0;
r_step[1] = r*Math.cos(3*Math.PI/8);
r_step[2] = r*Math.cos(Math.PI/4);
r_step[3] = r*Math.cos(Math.PI/8);
r_step[4] = r;
r_step[5] = r_step[3];
r_step[6] = r_step[2];
r_step[7] = r_step[1];
r_step[8] = 0;

var index = 0;
var height_step = 1;

/* The highest point of the upper part. */
wc[index++].setXYZ(0, r, 0);

for(var i=1; i<=7; i++)
{
    for(var k=0; k<8; k++)
    {
        var x = r_step[i]*cos[k];
        var y = h_step[height_step];
        var z = r_step[i]*sin[k];

        wc[index++].setXYZ(x, y, z);
    }
    height_step++;
}

/* The lowest point of the upper part. */
wc[index].setXYZ(0, -r, 0);
}

Sphere.prototype.assignProjectionsToFaces = function()
{
    var face = this.face;
    var p = this.projection;

    //Assign projections to triangles of the top of the sphere.
    for(var i=0; i<=6; i++)
        face[i] = [p[0], p[i+1], p[i+2]];

    face[7] = [p[0], p[8], p[1]];

    /* Assign projections to rectangular polygons. */
    for(var i=8; i<=55; i++)
    {

```

```

        if(((i + 1) % 8) == 0)
            face[i] = [p[i-7], p[i+1], p[i-6], p[i-14]];
        else
            face[i] = [p[i-7], p[i+1], p[i+2], p[i-6]];
    }

    /*
    * Assign projections to triangles of the bottom of the
    * sphere.
    */
    for(var i=56; i<=62; i++)
        face[i] = [p[i-7], p[57], p[i-6]];

    face[63] = [p[56], p[57], p[49]];
}

Sphere.prototype.createNormalVector = function()
{
    var p = this.plane;
    var n = this.avgNormalVectors;

    var v = this.projection
    /*
    * Find an average normal vector of vertex 0, the highest
    * point.
    */
    if(v[0].visibility == visible)
    {
        var adj1 = new Vector(p[0].A, p[0].B, p[0].C);
        var adj2 = new Vector(p[1].A, p[1].B, p[1].C);
        var adj3 = new Vector(p[2].A, p[2].B, p[2].C);
        var adj4 = new Vector(p[3].A, p[3].B, p[3].C);
        var adj5 = new Vector(p[4].A, p[4].B, p[4].C);
        var adj6 = new Vector(p[5].A, p[5].B, p[5].C);
        var adj7 = new Vector(p[6].A, p[6].B, p[6].C);
        n[0] = new Vector(p[7].A, p[7].B, p[7].C);
        n[0].addVector(adj1);
        n[0].addVector(adj2);
        n[0].addVector(adj3);
        n[0].addVector(adj4);
        n[0].addVector(adj5);
        n[0].addVector(adj6);
        n[0].addVector(adj7);
        n[0].normalize();
    }

    /* Find an average normal vector of vertex 1 to 56. */
    for(var i=1; i<=56; i++)
    {
        if(v[i].visibility == visible)
        {
            var adj1 = new Vector(p[i-1].A, p[i-1].B,
                                   p[i-1].C);
            var adj2 = new Vector(p[i+6].A, p[i+6].B,

```



```

        p[i+6].C);
    var adj3 = new Vector(p[i+7].A, p[i+7].B,
        p[i+7].C);
    if( (i - 1)%8 == 0)
        n[i] = new Vector(p[i+14].A,
            p[i+14].B, p[i+14].C);
    else
        n[i] = new Vector(p[i-2].A, p[i-2].B,
            p[i-2].C);

    n[i].addVector(adj1);
    n[i].addVector(adj2);
    n[i].addVector(adj3);
    n[i].normalize();
}
}

/* Find an average normal vector of vertex 57. */
if(v[57].visibility == visible)
{
    adj1 = new Vector(p[56].A, p[56].B, p[56].C);
    adj2 = new Vector(p[57].A, p[57].B, p[57].C);
    adj3 = new Vector(p[58].A, p[58].B, p[58].C);
    adj4 = new Vector(p[59].A, p[59].B, p[59].C);
    adj5 = new Vector(p[60].A, p[60].B, p[60].C);
    adj6 = new Vector(p[61].A, p[61].B, p[61].C);
    adj7 = new Vector(p[62].A, p[62].B, p[62].C);
    n[57] = new Vector(p[63].A, p[63].B, p[63].C);
    n[57].addVector(adj1);
    n[57].addVector(adj2);
    n[57].addVector(adj3);
    n[57].addVector(adj4);
    n[57].addVector(adj5);
    n[57].addVector(adj6);
    n[57].addVector(adj7);
    n[57].normalize();
}
}

Sphere.prototype.findColor = function()
{
    var avgI = this.avgI;
    var I = this.intensity;

    var p = this.projection;

    /* Calculate the average intensity of each edge. */
    if(p[0].visibility == visible)
    {
        for(var i=1; i<=8; i++)
        {
            if(p[i].visibility == visible)
                avgI[i-1] = (I[0] + I[i])/2;
        }
    }
}

```

```

    }
}

for(var i=1; i<= 48; i++)
{
    if(p[i].visibility == visible &&
        p[i+8].visibility == visible)
        avgI[i+7] = (I[i] + I[i+8])/2
}

if(p[57].visibility == visible)
{
    for(var i=49; i<=56; i++)
    {
        if(p[i].visibility == visible)
            avgI[i+7] = (I[57] + I[i])/2;
    }
}

/*
 * Get the RGB color of the Box and translate into HSV format.
 */
var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

var color_left = this.color_left;
var color_right = this.color_right;

var plane = this.plane;

/* Get a color on each edge in hsv format. */
for(var i=0; i<=63; i++)
{
    if(plane[i].visibility == visible)
    {
        /* Calculate color for left edges. */
        if(i%8 == 0)
            color_left[i] = hsv[2] - avgI[i]*(0.5);
        else if(i%8 != 0 && plane[i-1].visibility == visible)
            color_left[i] = color_right[i-1];
        else
            color_left[i] = hsv[2] - avgI[i]*(0.5);

        /* Calculate color for right edges. */
        if((i+1)%8 == 0 && plane[i-7].visibility == visible)
            color_right[i] = color_left[i-7];
        else if((i+1)%8 == 0 &&
            plane[i-7].visibility == invisible)
            color_right[i] = hsv[2] - avgI[i-7]*(0.5);
        else
            color_right[i] = hsv[2] - avgI[i+1]*(0.5);
    }
}
}

```

```

Sphere.prototype.modify = function()
{
    this.createVC();           /* super class's method */
    this.createProjection();   /* super class's method */
    this.assignProjectionsToFaces();
    this.adjustCenter()       /* super class's method */
    this.createPlane();        /* super class's method */
    this.createNormalVector();
    this.findIntensity();      /* super class's method */
    this.findColor();
    this.modifyGradientTag();  /* super class's method */
}

Sphere.prototype.getVCandProjection = function()
{
    this.createVC();           /* super class's method */
    this.createProjection();   /* super class's method */
    this.assignProjectionsToFaces();
    this.adjustCenter()       /* super class's method */
    this.createPlane();        /* super class's method */
}

Sphere.prototype.getShade = function()
{
    this.createNormalVector();
    this.findIntensity();      /* super class's method */
    this.findColor();
    this.modifyGradientTag();  /* super class's method */
}

/*****
/*      Cylinder Class, a subclass of Primitive
* All methods of this class are performing the same function as Box's
* methods do but purposely for a Cylinder object. So the way we model
* world coordinates and assign them to each surface is different from
* the other kinds of objects. However, the concept of calculating
* viewing coordinates and color intensities remains the same.
*/
*****/

Cylinder.prototype = new Primitive();
Cylinder.prototype.constructor = Cylinder;
Cylinder.superclass = Primitive.prototype;

function Cylinder(thisNode, appearanceNode, id)
{
    if(arguments.length > 0)
        this.init(thisNode, appearanceNode, id);
}

Cylinder.prototype.init = function(thisNode, appearanceNode, id)
{
    /* Call a superclass init. */
    Cylinder.superclass.init.call(this, thisNode, appearanceNode,
        id);
}

```

```

        this.radius = thisNode.getAttribute("radius");
        this.height = thisNode.getAttribute("height");

        this.createFirstWC();
        this.modifyWC();          /* super class's method */
    }

Cylinder.prototype.createFirstWC = function()
{
    var wc = this.wc;
    var r = this.radius;
    var h = this.height;

    var cos = new Array(8);  /* Store the cos data. */
    cos[0] = 1;
    cos[1] = Math.cos(Math.PI/4);
    cos[2] = 0;
    cos[3] = -cos[1];
    cos[4] = -1;
    cos[5] = -cos[1];
    cos[6] = 0;
    cos[7] = cos[1];

    var sin = new Array(8);  /* Store the sin data. */
    sin[0] = 0;
    sin[1] = -cos[1];        /* cos(pi/4) == sin(pi/4) */
    sin[2] = -1;
    sin[3] = sin [1];
    sin[4] = 0;
    sin[5] = -sin[1];
    sin[6] = 1;
    sin[7] = -sin[1];

    for(var i=0; i<8; i++)
    {
        var x = r*cos[i];
        var y = h/2;
        var z = r*sin[i];

        wc[i].setXYZ(x, y, z);
    }

    for(var i=8; i<16; i++)
    {
        var x = r*cos[i-8];
        var y = -h/2;
        var z = r*sin[i-8];

        wc[i].setXYZ(x, y, z);
    }
}

Cylinder.prototype.assignProjectionsToFaces = function()

```

```

{
    var face = this.face;
    var p = this.projection;

    /* Side faces of a cylinder. */
    for(var i=0; i<=6; i++)
        face[i] = [p[i], p[i+8], p[i+9], p[i+1]];

    face[7] = [p[7], p[15], p[8], p[0]];

    /* A top */
    face[8] = [p[0], p[1], p[2], p[3], p[4], p[5], p[6], p[7]];
    /* A bottom */
    face[9] = [ p[15], p[14], p[13], p[12], p[11], p[10], p[9],
        p[8]];
}

Cylinder.prototype.createNormalVector = function()
{
    var p = this.plane;
    var n = this.avgNormalVectors;
    var v = this.projection;

    /*
    * An average normal vector at vertex 0 is an average vector
    * of plane 0 and 7.
    */
    if(v[0].visibility == visible)
    {
        var adj1 = new Vector(p[7].A, p[7].B, p[7].C);
        n[0] = new Vector(p[0].A, p[0].B, p[0].C);
        n[0].addVector(adj1);
        n[0].normalize();

        n[8] = new Vector(n[0].A, n[0].B, n[0].C);
    }
    else if(v[0].visibility == invisible && v[8].visibility ==
        visible)
    {
        var adj1 = new Vector(p[7].A, p[7].B, p[7].C);
        n[8] = new Vector(p[0].A, p[0].B, p[0].C);
        n[8].addVector(adj1);
        n[8].normalize();
    }

    /* Find an average normal vector. */
    for(var i=1; i<=7; i++)
    {
        if(v[i].visibility == visible)
        {
            var adj1 = new Vector(p[i-1].A, p[i-1].B, p[i-1].C);
            n[i] = new Vector(p[i].A, p[i].B, p[i].C);
            n[i].addVector(adj1);
        }
    }
}

```

```

        n[i].normalize();

        n[i + 8] = new Vector(n[i].A, n[i].B, n[i].C);
    }
    else if(v[i].visibility == invisible &&
        v[i + 8].visibility == visible)
    {
        var adj1 = new Vector(p[i-1].A, p[i-1].B, p[i-1].C);
        n[i + 8] = new Vector(p[i].A, p[i].B, p[i].C);
        n[i + 8].addVector(adj1);
        n[i + 8].normalize();
    }
}

Cylinder.prototype.findColor = function()
{
    var I = this.intensity;

    /*
    * Get the RGB color of the Box and translate into HSV format.
    */
    var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

    var color_left = this.color_left;
    var color_right = this.color_right;

    var p = this.plane;

    /* Get a color on each edge in hsv format. */
    for(var i=0; i<8; i++)
    {
        if(p[i].visibility == visible)
        {
            if(i%8 == 0)
                color_left[i] = hsv[2] - I[i]*(0.5);
            else if(p[i - 1].visibility == visible)
                color_left[i] = color_right[i-1];
            else
                color_left[i] = hsv[2] - I[i]*(0.5);

            if((i+1)%8 == 0 && p[i-7].visibility == visible)
                color_right[i] = color_left[i-7];
            else if((i+1)%8 == 0 &&
                p[i-7].visibility == invisible)
                color_right[i] = hsv[2] - I[i - 7]*(0.5);
            else
                color_right[i] = hsv[2] - I[i+1]*(0.5);
        }
    }

    /* Get a color for the top. */
    color_left[8] = hsv[2] - I[4]*(0.5);
    color_right[8] = hsv[2] - I[0]*(0.5);

```

```

        /* Get a color for the bottom. */
        color_left[9] = hsv[2] - I[12]*(0.5);
        color_right[9] = hsv[2] - I[8]*(0.5);
    }

    Cylinder.prototype.modify = function()
    {
        this.createVC();           /* super class's method */
        this.createProjection();    /* super class's method */
        this.assignProjectionsToFaces();
        this.adjustCenter()        /* super class's method */
        this.createPlane();        /* super class's method */
        this.createNormalVector();
        this.findIntensity();      /* super class's method */
        this.findColor();
        this.modifyGradientTag();  /* super class's method */
    }

    Cylinder.prototype.getVCandProjection = function()
    {
        this.createVC();           /* super class's method */
        this.createProjection();    /* super class's method */
        this.assignProjectionsToFaces();
        this.adjustCenter()        /* super class's method */
        this.createPlane();        /* super class's method */
    }

    Cylinder.prototype.getShade = function()
    {
        this.createNormalVector();
        this.findIntensity();      /* super class's method */
        this.findColor();
        this.modifyGradientTag();  /* super class's method */
    }
}
/*****
/*
* This function is to return a node that has a value of the
* "DEF" attribute equal to the specified name.
* @para group is the target group nodes and name is the
* specified name we are looking for.
* @return a group node whose "DEF" value is the specified name
* or null if no associated group node has been found.
*/
function getNodeFromGroup(group, name)
{
    for(var i=0; i<group.length; i++)
    {
        if(group[i].getAttribute("DEF") == name)
            return group[i];
    }

    return null;
}

```

```

/*
 * This function is to clone a new Element including its
 * descendent so that we can manipulate this new element
 * arbitrarily.
 * @para node is the prototype which we want to copy from it.
 * @return a clone node.
 */
function cloneElement(node)
{
    var attr = node.attributes;
    var len = attr.length;
    var name = node.nodeName;

    var element = document.createElement(name);
    var attributes = new Array(len);
    for(var i=0; i<len; i++)
    {
        attributes[i] = document.createAttribute(attr[i].name);
        attributes[i].value = attr[i].value;
        element.attributes.setNamedItem(attributes[i]);
    }

    if(node.hasChildNodes())
    {
        var children = node.childNodes;
        var len1 = children.length;
        var cloneChild = new Array(len1);
        for(var i=0; i<len1; i++)
        {
            cloneChild[i] = cloneElement(children[i]);
            element.appendChild(cloneChild[i]);
        }

        return element;
    }
    else
        return element;
}

/*
 * This function will modify a DOM structure of a source file to
 * be ready for further translation tasks. We will replace
 * <Group> tags which refer to another <Group> tag with the
 * associated <Group> tag along with its descendent.
 */
function transformX3D()
{
    var groupDEF = new Array();
    var countDEF = 0;
    var groupUSE = new Array();
    var countUSE = 0;
    /*
     * Separate group nodes into two groups, groupDEF and

```



```

    * groupUSE.
    */
    for(var i=0; i<g_groupNodes.length; i++)
    {
        if(g_groupNodes[i].getAttribute("DEF") != null)
            groupDEF[countDEF++] = g_groupNodes[i];
        else if(g_groupNodes[i].getAttribute("USE") != null)
            groupUSE[countUSE++] = g_groupNodes[i];
    }

    for(var i=0; i<groupUSE.length; i++)
    {
        var targetName = groupUSE[i].getAttribute("USE");

        // Search a group whose DEF value is equal to targetName
        var targetGroup = getNodeFromGroup(groupDEF, targetName);
        var dup = targetGroup.cloneNode(true);
        var parent = groupUSE[i].parentNode;

        /* replace this node with the targetGroup. */
        parent.replaceChild(dup, groupUSE[i]);
    }
}

/*
 * This function will be called after the HTML is loaded. This
 * function is to create, not render yet, primitive objects of
 * this scene and keep them as global variables, so they can be
 * used later in function drawObjects to render 3D objects on the
 * scene.
 */
function buildSVG()
{
    /* Append an SVG root as a child of the body node. */
    g_bodyNode.appendChild(g_svg);

    /*
     * Replace a calling <Group> node with its associated <Group>
     * node and its descendants.
     */
    transformX3D();

    var len = g_shapeNodes.length;

    for(var i=0; i<len; i++)
    {
        var temp =
            g_shapeNodes[i].getElementsByTagName("Box");
        var aBox = temp[0];
        temp = g_shapeNodes[i].getElementsByTagName("Cone");
        var aCone = temp[0];
        temp = g_shapeNodes[i].getElementsByTagName("Cylinder");
        var aCylinder = temp[0];
        temp = g_shapeNodes[i].getElementsByTagName("Sphere");
    }
}

```

```

var aSphere = temp[0];
temp = g_shapeNodes[i].getElementsByTagName("Text");

var aText = temp[0];

temp = g_shapeNodes[i].getElementsByTagName("Appearance");
var Appearance = temp[0];

if(aBox != null)
{
    /* Create a box object. */
    var box = new Box(aBox, Appearance, i);

    /* Keep a new box as global object. */
    g_objects[i] = box;
}
else if(aCone != null)
{
    /* Create a cone object. */
    var cone = new Cone(aCone, Appearance, i);

    /* Keep a new cone as global object. */
    g_objects[i] = cone;
}
else if(aSphere != null)
{
    /* Create a sphere object. */
    var sphere = new Sphere(aSphere, Appearance, i);

    /* Keep a new sphere as global object. */
    g_objects[i] = sphere;
}
else if(aCylinder != null)
{
    /* Create a cylinder object. */
    var cylinder = new Cylinder(aCylinder,
        Appearance, i);

    /* Keep a new cylinder as global object. */
    g_objects[i] = cylinder;
}
}
/* Render every objects of this scene. */
drawObjects();
}

//This function renders every objects created by buildSVG( ).
function drawObjects()
{
    var count = 0;

    // Create a transformation matrix of a current perspective.

```

```

perspective.translate(g_tx, g_ty, g_tz);
perspective.rotate((g_rx % 360)*0.017453293,
    (g_ry % 360)*0.017453293, (g_rz % 360)*0.017453293);

/*
 * Get viewing coordinates and projections of this current
 * perspective.
 */
var obj_len = g_objects.length;

for(var i=0; i<obj_len; i++)
    g_objects[i].getVCandProjection();

/*
 * Arrange the order of drawing objects according to the depth
 * of their center.
 */
quicksortCenter(g_objects, 0, obj_len - 1);

var countVisible = 0;
var countInvisible = 0;
var visiblePolygons = new Array();
var invisiblePolygons = new Array();

for(var i=0; i<obj_len; i++)
{
    var vertices = new Array(4);
    var a_polygon = new Array();
    for(var k=0; k<g_objects[i].num_faces; k++)
    {
        vertices = g_objects[i].face[k];
        a_polygon[k] = new Polygon(vertices, k,
            g_objects[i], g_objects[i].plane[k]);
    }

    var a_polygon_len = a_polygon.length;

    /*
     * Separate polygons into two groups, visible and
     * invisible polygons.
     */
    var start_vi = countVisible;
    for(var k=0; k<a_polygon_len; k++)
    {
        /* Check if this polygon is visible. */
        if(a_polygon[k].isVisible())
        {
            a_polygon[k].setVisibility(visible);
            visiblePolygons[countVisible++] = a_polygon[k];
        }
        else
            invisiblePolygons[countInvisible++] = a_polygon[k];
    }
}

```

```

        end = countVisible - 1;
        /* Arrange the rendering order of visible polygons.*/
        quicksortDepth(visiblePolygons, start_vi, end);
    }

    /* Calculate color intensity. */
    for(var i=0; i<obj_len; i++)
        g_objects[i].getShade();

    var invi_len = invisiblePolygons.length;
    var vi_len = visiblePolygons.length;
    var tag_len = g_polygonSVGTag.length;
    var start_paint = tag_len - vi_len;

    for(var i=0; i<start_paint; i++)
    {
        /* Clear data in global svg polygon tags. */
        g_polygonSVGTag[i].setAttribute("points", "0,0");
        g_polygonSVGTag[i].setAttribute("fill", "none");
        g_polygonSVGTag[i].setAttribute("stroke", "none");
    }

    /* Render visible polygons. */
    for(var i=0; i<vi_len; i++)
    {
        /*
        * Clear contents of the TAG which might left from the
        * previous perspective.
        */
        g_polygonSVGTag[i + start_paint].setAttribute("points",
            "0,0");
        g_polygonSVGTag[i + start_paint].setAttribute("fill",
            "none");
        g_polygonSVGTag[i + start_paint].setAttribute("stroke",
            "none");

        var projection = visiblePolygons[i].vertices;
        var points = visiblePolygons[i].stringOfPoints;
        g_polygonSVGTag[i + start_paint].setAttribute("points",
            points);

        /* Apply shading on the polygon surface. */
        var url = "url(#" + visiblePolygons[i].object.id +
            visiblePolygons[i].id + ")";
        g_polygonSVGTag[i + start_paint].setAttribute("fill",
            url);
    }

    /* Reset the perspective matrix to be an identity matrix. */
    perspective.reset();
}
</script>
</html>

```

## Code for the translator version 1

Only a few functions in version 1 and version 2 are different. Instead of show a whole program which is mostly similar to each other, we only show the functions in version 1 that are different from those in version 2.

```
function Point3D(x, y, z) /* Constructor */
{
    this.x3D = x;
    this.y3D = y;
    this.z3D = z;
    this.h = 1;

    /* xp, yp, and zp are projection coordinates of this point. */
    this.xp = 0;
    this.yp = 0;
    this.zp = 0;
    this.h = 0;
}

function Polygon(vertices, id, object, plane) /* Constructor */
{
    this.vertices = vertices;
    this.id = id;
    this.object = object; /* The object this polygon belongs to. */
    this.plane = plane;

    /*
    * stringOfPoints is containing projected positions in string format
    * which will be later supplied to a "points" attribute of a
    * <polygon> tag.
    */
    var len = vertices.length;
    if(len == 8)
        this.stringOfPoints = "" + vertices[0].xp + ", " +
            vertices[0].yp + ", " + vertices[1].xp + ", " +
            vertices[1].yp + ", " + vertices[2].xp + ", " +
            vertices[2].yp + ", " + vertices[3].xp + ", " +
            vertices[3].yp + ", " + vertices[4].xp + ", " +
            vertices[4].yp + ", " + vertices[5].xp + ", " +
            vertices[5].yp + ", " + vertices[6].xp + ", " +
            vertices[6].yp + ", " + vertices[7].xp + ", " +
            vertices[7].yp;
    else if(len == 4)
        this.stringOfPoints = "" + vertices[0].xp + ", " +
            vertices[0].yp + ", " + vertices[1].xp + ", " +
            vertices[1].yp + ", " + vertices[2].xp + ", " +
```

```

        vertices[2].yp + ", " + vertices[3].xp + ", " +
        vertices[3].yp;
    else if(len == 3)
        this.stringOfPoints = "" + vertices[0].xp + ", " +
        vertices[0].yp + ", " + vertices[1].xp + ", " +
        vertices[1].yp + ", " + vertices[2].xp + ", " +
        vertices[2].yp;

    /* Find the depth of this polygon. */
    this.depth = this.getDepth();
}

function Plane(P1, P2, P3)    /* Constructor */
{
    var x1 = P1.x3D;
    var x2 = P2.x3D;
    var x3 = P3.x3D;

    var y1 = P1.y3D;
    var y2 = P2.y3D;
    var y3 = P3.y3D;

    var z1 = P1.z3D;
    var z2 = P2.z3D;
    var z3 = P3.z3D;

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

/* This method is to find the intensity of each vertex. */
Primitive.prototype.findIntensity = function()
{
    var centerPoint = this.center2;
    var l = new Vector(centerPoint.x3D, centerPoint.y3D,
        centerPoint.z3D - g_viewPoint);
    l.normalize();
    var h = new Vector(l.A, l.B, l.C);
    h.addVector(l);
    h.normalize();

    var n = this.avgNormalVectors;
    var I = this.intensity;
    var pa = 0.2;
    var pd = 0.6;
    var ps = 0.2;

    var p = this.projection;
    /* Calculate the intensity of each vertex. */
    for(var i=0; i<this.num_vertices; i++)
    {

```

```

        // Calculate only light intensity of a visible polygon vertex.
        var Id = new Vector(l.A, l.B, l.C);
        l_dot_n = Id.dotVector(n[i]);
        var Is = new Vector(h.A, h.B, h.C);
        h_dot_n = Is.dotVector(n[i]);
        I[i] = pa + pd*l_dot_n + ps*(h_dot_n * h_dot_n);
    }
}

Primitive.prototype.modifyGradientTag = function()
{
    var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);
    var p = this.plane;

    for(var i=0; i<this.num_faces; i++)
    {
        var shadeId = "" + this.id + i;
        var shadeTag =
            document.getElementById(shadeId);
        var leftStop = shadeTag.firstChild;
        var rightStop = shadeTag.lastChild;

        /* Get the rgb color on each edge of a polygon. */
        var rgb_left = hsvTorgb(hsv[0], hsv[1], this.color_left[i]);
        var rgb_right = hsvTorgb(hsv[0], hsv[1], this.color_right[i]);

        // Change the format to be sRGB which is legally used in SVG.
        rgb_left[0] *= 255; rgb_left[1] *= 255; rgb_left[2] *= 255;
        rgb_right[0] *= 255; rgb_right[1] *= 255; rgb_right[2] *= 255;

        var c_l = "rgb(" + rgb_left[0] + ", " + rgb_left[1] + ", " +
            rgb_left[2] + ")";
        var c_r = "rgb(" + rgb_right[0] + ", " + rgb_right[1] + ", " +
            rgb_right[2] + ")";

        /* Supply stop colors to a <linearGradient> tag. */
        leftStop.setAttribute("stop-color", c_l);
        rightStop.setAttribute("stop-color", c_r);
    }
}

Box.prototype.createNormalVector = function()
{
    var p = this.plane;
    var n = this.avgNormalVectors;

    var front = 0; var back = 1; var left = 2;
    var right = 3; var top = 4; var bottom = 5;

    var f = new Vector(p[front].A, p[front].B, p[front].C);
    var b = new Vector(p[back].A, p[back].B, p[back].C);
    var l = new Vector(p[left].A, p[left].B, p[left].C);
    var r = new Vector(p[right].A, p[right].B, p[right].C);

```

```

var t = new Vector(p[top].A, p[top].B, p[top].C);
var butt = new Vector(p[bottom].A, p[bottom].B, p[bottom].C);

/*
 * Create an average normal vector of corner 0 which is an average
 * combination of a front, top, and left face.
 */
n[0] = new Vector(p[front].A, p[front].B, p[front].C);
n[0].addVector(t);
n[0].addVector(l);
n[0].normalize();

/*
 * Create an average normal vector of corner 1 which is an average
 * combination of a front, top, and right plane.
 */
n[1] = new Vector(p[front].A, p[front].B, p[front].C);
n[1].addVector(t);
n[1].addVector(r);
n[1].normalize();

/*
 * Create an average normal vector of corner 2 which is an average
 * combination of a front, back, and right plane.
 */
n[2] = new Vector(p[front].A, p[front].B, p[front].C);
n[2].addVector(b);
n[2].addVector(r);
n[2].normalize();

/*
 * Create an average normal vector of corner 3 which is an average
 * combination of a front, back, and left plane.
 */
n[3] = new Vector(p[front].A, p[front].B, p[front].C);
n[3].addVector(b);
n[3].addVector(l);
n[3].normalize();

/*
 * Create an average normal vector of corner 4 which is an average
 * combination of a back, top, and left plane.
 */
n[4] = new Vector(p[back].A, p[back].B, p[back].C);
n[4].addVector(t);
n[4].addVector(l);
n[4].normalize();

/*
 * Create an average normal vector of corner 5 which is an average
 * combination of a back, top, and right plane.
 */
n[5] = new Vector(p[back].A, p[back].B, p[back].C);
n[5].addVector(t);

```



```

n[5].addVector(r);
n[5].normalize();

/*
 * Create an average normal vector of corner 6 which is an average
 * combination of a back, bottom, and right plane.
 */
n[6] = new Vector(p[back].A, p[back].B, p[back].C);
n[6].addVector(butt);
n[6].addVector(r);
n[6].normalize();

/*
 * Create an average normal vector of corner 7 which is an average
 * combination of a back, bottom, and left plane.
 */
n[7] = new Vector(p[back].A, p[back].B, p[back].C);
n[7].addVector(butt);
n[7].addVector(l);
n[7].normalize();
}

Box.prototype.findColor = function()
{
    var avgI = this.avgI;
    var I = this.intensity;

    /* Calculate the average intensity of each edge. */
    avgI[0] = (I[0] + I[3])/2; avgI[1] = (I[1] + I[2])/2;
    avgI[2] = (I[5] + I[6])/2; avgI[3] = (I[4] + I[7])/2;
    avgI[4] = (I[4] + I[0])/2; avgI[5] = (I[5] + I[1])/2;
    avgI[6] = (I[3] + I[7])/2; avgI[7] = (I[2] + I[6])/2;

    /* Get the rgb color and translate into hsv format. */
    var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

    var front = 0; var back = 1; var left = 2; var right = 3;
    var top = 4; var bottom = 5;
    var color_left = this.color_left;
    var color_right = this.color_right;

    /* Colors on each edge are in hsv format. */
    color_left[front] = hsv[2] - avgI[0]*(0.5);
    color_right[front] = hsv[2] - avgI[1]*(0.5);
    color_left[back] = hsv[2] - avgI[2]*(0.5);
    color_right[back] = hsv[2] - avgI[3]*(0.5);
    color_left[left] = hsv[2] - avgI[3]*(0.5);
    color_right[left] = hsv[2] - avgI[0]*(0.5);
    color_left[right] = hsv[2] - avgI[1]*(0.5);
    color_right[right] = hsv[2] - avgI[2]*(0.5);
    color_left[top] = hsv[2] - avgI[4]*(0.5);
    color_right[top] = hsv[2] - avgI[5]*(0.5);
    color_left[bottom] = hsv[2] - avgI[6]*(0.5);
    color_right[bottom] = hsv[2] - avgI[7]*(0.5);

```

```

}

Cone.prototype.createNormalVector = function()
{
    var p = this.plane;
    var n = this.avgNormalVectors;
    /*
    * An average normal vector at vertex 0 is an average vector of plane
    * 0 and 7.
    */
    var adj1 = new Vector(p[7].A, p[7].B, p[7].C);
    n[0] = new Vector(p[0].A, p[0].B, p[0].C);
    n[0].addVector(adj1);
    n[0].normalize();

    /* Find an average normal vector of vertex 1 to 7. */
    for(var i=1; i<=7; i++)
    {
        var adj1 = new Vector(p[i-1].A, p[i-1].B, p[i-1].C);
        n[i] = new Vector(p[i].A, p[i].B, p[i].C);
        n[i].addVector(adj1);
        n[i].normalize();
    }

    /* Find an average normal vector of vertex 8 to 61. */
    for(var i=8; i<=31; i++)
    {
        var adj1 = new Vector(p[i-8].A, p[i-8].B, p[i-8].C);
        var adj2 = new Vector(p[i-1].A, p[i-1].B, p[i-1].C);
        var adj3 = "";

        if( i%8 == 0)
            adj3 = new Vector(p[i+7].A, p[i+7].B, p[i+7].C);

        else
            adj3 = new Vector(p[i-9].A, p[i-9].B, p[i-9].C)

        n[i] = new Vector(p[i].A, p[i].B, p[i].C);
        n[i].addVector(adj1);
        n[i].addVector(adj2);
        n[i].addVector(adj3);
        n[i].normalize();
    }

    /* Find an average normal vector of vertex 32. */
    var adj1 = new Vector(p[24].A, p[24].B, p[24].C);
    var adj2 = new Vector(p[25].A, p[25].B, p[25].C);
    var adj3 = new Vector(p[26].A, p[26].B, p[26].C);
    var adj4 = new Vector(p[27].A, p[27].B, p[27].C);
    var adj5 = new Vector(p[28].A, p[28].B, p[28].C);
    var adj6 = new Vector(p[29].A, p[29].B, p[29].C);
    var adj7 = new Vector(p[30].A, p[30].B, p[30].C);
    n[32] = new Vector(p[31].A, p[31].B, p[31].C);
    n[32].addVector(adj1);

```

```

    n[32].addVector(adj2);
    n[32].addVector(adj3);
    n[32].addVector(adj4);
    n[32].addVector(adj5);
    n[32].addVector(adj6);
    n[32].addVector(adj7);
    n[32].normalize();
}

Cone.prototype.findColor = function()
{
    var avgI = this.avgI;
    var I = this.intensity;

    /* Calculate the average intensity of each edge. */
    for(var i=0; i<=23; i++)
        avgI[i] = (I[i] + I[i+8])/2;

    for(var i=24; i<=31; i++)
        avgI[i] = (I[i] + I[32])/2;

    /* Get the RGB color of the Box and translate into HSV format. */
    var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

    var color_left = this.color_left;
    var color_right = this.color_right;

    /* Get a color on each edge in hsv format. */
    for(var i=0; i<=31; i++)
    {
        if(i%8 == 0)
            color_left[i] = hsv[2] - avgI[i]*(0.5);
        else
            color_left[i] = color_right[i-1];

        if((i+1)%8 == 0)
            color_right[i] = color_left[i-7];
        else
            color_right[i] = hsv[2] - avgI[i+1]*(0.5);
    }

    /* Get a color for the bottom of the cone. */
    color_left[32] = hsv[2] - I[4]*(0.5);
    color_right[32] = hsv[2] - I[0]*(0.5);
}

Sphere.prototype.createNormalVector = function()
{
    var p = this.plane;
    var n = this.avgNormalVectors;

    /* Find an average normal vector of vertex 0, the highest point. */
    var adj1 = new Vector(p[0].A, p[0].B, p[0].C);
    var adj2 = new Vector(p[1].A, p[1].B, p[1].C);

```

```

var adj3 = new Vector(p[2].A, p[2].B, p[2].C);
var adj4 = new Vector(p[3].A, p[3].B, p[3].C);
var adj5 = new Vector(p[4].A, p[4].B, p[4].C);
var adj6 = new Vector(p[5].A, p[5].B, p[5].C);
var adj7 = new Vector(p[6].A, p[6].B, p[6].C);
n[0] = new Vector(p[7].A, p[7].B, p[7].C);
n[0].addVector(adj1);
n[0].addVector(adj2);
n[0].addVector(adj3);
n[0].addVector(adj4);
n[0].addVector(adj5);
n[0].addVector(adj6);
n[0].addVector(adj7);
n[0].normalize();

/* Find an average normal vector of vertex 1 to 56. */
for(var i=1; i<=56; i++)
{
    var adj1 = new Vector(p[i-1].A, p[i-1].B, p[i-1].C);
    var adj2 = new Vector(p[i+6].A, p[i+6].B, p[i+6].C);
    var adj3 = new Vector(p[i+7].A, p[i+7].B, p[i+7].C);

    if( (i - 1)%8 == 0)
        n[i] = new Vector(p[i+14].A, p[i+14].B, p[i+14].C);
    else
        n[i] = new Vector(p[i-2].A, p[i-2].B, p[i-2].C);

    n[i].addVector(adj1);
    n[i].addVector(adj2);
    n[i].addVector(adj3);
    n[i].normalize();
}

/* Find an average normal vector of vertex 57, the lowest point. */
adj1 = new Vector(p[56].A, p[56].B, p[56].C);
adj2 = new Vector(p[57].A, p[57].B, p[57].C);
adj3 = new Vector(p[58].A, p[58].B, p[58].C);
adj4 = new Vector(p[59].A, p[59].B, p[59].C);
adj5 = new Vector(p[60].A, p[60].B, p[60].C);
adj6 = new Vector(p[61].A, p[61].B, p[61].C);
adj7 = new Vector(p[62].A, p[62].B, p[62].C);
n[57] = new Vector(p[63].A, p[63].B, p[63].C);
n[57].addVector(adj1);
n[57].addVector(adj2);
n[57].addVector(adj3);
n[57].addVector(adj4);
n[57].addVector(adj5);
n[57].addVector(adj6);
n[57].addVector(adj7);
n[57].normalize();
}

Sphere.prototype.findColor = function()
{

```

```

var avgI = this.avgI;
var I = this.intensity;

/* Calculate the average intensity of each edge. */
for(var i=0; i<=7; i++)
    avgI[i] = (I[0] + I[i+1])/2;
for(var i=8; i<=55; i++)
    avgI[i] = (I[i-7] + I[i+1])/2;
for(var i=56; i<=63; i++)
    avgI[i] = (I[57] + I[i-7])/2;

/* Get the RGB color of the Box and translate into HSV format. */
var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

var color_left = this.color_left;
var color_right = this.color_right;

/* Get a color on each edge in hsv format. */
for(var i=0; i<=63; i++)
{
    if(i%8 == 0)
        color_left[i] = hsv[2] - avgI[i]*(0.5);
    else
        color_left[i] = color_right[i-1];

    if((i+1)%8 == 0)
        color_right[i] = color_left[i-7];
    else
        color_right[i] = hsv[2] - avgI[i+1]*(0.5);
}
}

Cylinder.prototype.createNormalVector = function()
{
    var p = this.plane;
    var n = this.avgNormalVectors;

    /*
    * An average normal vector at vertex 0 is an average vector of plane
    * 0 and 7.
    */
    var adj1 = new Vector(p[7].A, p[7].B, p[7].C);
    n[0] = new Vector(p[0].A, p[0].B, p[0].C);
    n[0].addVector(adj1);
    n[0].normalize();

    n[8] = new Vector(n[0].A, n[0].B, n[0].C);

    /* Find an average normal vector. */
    for(var i=1; i<=7; i++)
    {
        var adj1 = new Vector(p[i-1].A, p[i-1].B, p[i-1].C);
        n[i] = new Vector(p[i].A, p[i].B, p[i].C);
        n[i].addVector(adj1);
    }
}

```

```

        n[i].normalize();

        n[i + 8] = new Vector(n[i].A, n[i].B, n[i].C);
    }
}

Cylinder.prototype.findColor = function()
{
    var I = this.intensity;
    /* Get the RGB color of the Box and translate into HSV format. */
    var hsv = rgbTohsv(this.rgb[0], this.rgb[1], this.rgb[2]);

    var color_left = this.color_left;
    var color_right = this.color_right;

    /* Get a color on each edge in hsv format. */
    for(var i=0; i<8; i++)
    {
        if(i%8 == 0)
            color_left[i] = hsv[2] - I[i]*(0.5);
        else
            color_left[i] = color_right[i-1];

        if((i+1)%8 == 0)
            color_right[i] = color_left[i-7];
        else
            color_right[i] = hsv[2] - I[i+1]*(0.5);
    }

    /* Get a color for the top. */
    color_left[8] = hsv[2] - I[4]*(0.5);
    color_right[8] = hsv[2] - I[0]*(0.5);

    /* Get a color for the bottom. */
    color_left[9] = hsv[2] - I[12]*(0.5);
    color_right[9] = hsv[2] - I[8]*(0.5);
}

/* This function is to render every objects created by buildSVG( ). */
function drawObjects()
{
    var count = 0;

    /* Create a transformation matrix of a current perspective. */
    perspective.translate(g_tx, g_ty, g_tz);
    perspective.rotate((g_rx % 360)*0.017453293,
        (g_ry % 360)*0.017453293, (g_rz % 360)*0.017453293);

    /*
    * Get viewing coordinates and projections of this current
    * perspective.
    */
    var obj_len = g_objects.length;

```

```

for(var i=0; i<obj_len; i++)
{
    g_objects[i].getVCandProjection();
    g_objects[i].getShade();
}

/*
 * Arrange the order of drawing objects according to the depth of
 * their center.
 */
quicksortCenter(g_objects, 0, obj_len - 1);

var countVisible = 0;
var countInvisible = 0;
var visiblePolygons = new Array();
var invisiblePolygons = new Array();

for(var i=0; i<obj_len; i++)
{
    var vertices = new Array(4);
    var a_polygon = new Array();
    for(var k=0; k<g_objects[i].num_faces; k++)
    {
        vertices = g_objects[i].face[k];
        a_polygon[k] = new Polygon(vertices, k, g_objects[i],
            g_objects[i].plane[k]);
    }

    var a_polygon_len = a_polygon.length;

    /*
     * Separate polygons into two groups, visible and invisible
     * polygons.
     */
    var start_vi = countVisible;
    for(var k=0; k<a_polygon_len; k++)
    {
        /* Check if this polygon is visible. */
        if(a_polygon[k].isVisible())
            visiblePolygons[countVisible++] = a_polygon[k];
        else
            invisiblePolygons[countInvisible++] = a_polygon[k];
    }
    end = countVisible - 1;
    quicksortDepth(visiblePolygons, start_vi, end);
}

// Draw invisible polygons before other visible polygons.
var invi_len = invisiblePolygons.length;
var vi_len = visiblePolygons.length;

var total = invi_len + vi_len;

for(var i=0; i<invi_len; i++)

```

```

{
    /* Clear data in global svg polygon tags. */
    polygonSVGTag[i].setAttribute("points", "0,0");
    polygonSVGTag[i].setAttribute("fill", "none");
    polygonSVGTag[i].setAttribute("stroke", "none");

    var projection = invisiblePolygons[i].vertices;
    var points = invisiblePolygons[i].stringOfPoints;
    polygonSVGTag[i].setAttribute("points", points);

    /* Apply shading on the polygon surface. */
    var url = "url(#" + invisiblePolygons[i].object.id +
        invisiblePolygons[i].id + ")";
    polygonSVGTag[i].setAttribute("fill", url);
}

for(var i=0; i<vi_len; i++)
{
    /* Clear data in global svg polygon tags. */
    polygonSVGTag[i + invi_len].setAttribute("points", "0,0");
    polygonSVGTag[i + invi_len].setAttribute("fill", "none");
    polygonSVGTag[i + invi_len].setAttribute("stroke", "none");

    var projection = visiblePolygons[i].vertices;
    var points = visiblePolygons[i].stringOfPoints;
    polygonSVGTag[i + invi_len].setAttribute("points", points);

    /* Apply shading on the polygon surface. */
    var url = "url(#" + visiblePolygons[i].object.id +
        visiblePolygons[i].id + ")";
    polygonSVGTag[i + invi_len].setAttribute("fill", url);
}

/* Reset the perspective matrix to be an identity matrix. */
perspective.reset();
}

```

### Code for the translator version 3

Only a few functions in version 3 and version 2 are different. Instead of show a whole program which is mostly similar to each other, we only show the functions in version 3 that are different from those in version 2.

```

/*
 * This function is to remove the hidden front-face polygon from an
 * array of visible polygons. And add them into invisible polygons, so
 * we can reduce the computation time when calculating shading.
 */
function removeHiddenSurface(visibleSurfaces, invisibleSurfaces)
{

```



```

/*
 * Check if each surface is hidden by other surfaces of different
 * objects. If so, it can be removed. Start to check the second
 * narrowest surface; the narrowest one is definitely not occluded by
 * any surface.
 */
var len = visibleSurfaces.length;
for(var i=len - 2; i>=0; i--)
{
    visibleSurfaces[i].resetHiddenProperty();
    var thisObjId = visibleSurfaces[i].object.id;
    var vertices = visibleSurfaces[i].vertices;

    /*
     * Comparing this surface with every other closer front-face
     * surfaces of different objects to see if this surface is hidden
     * by any surfaces. If so, we can remove this surface.
     */
    for(var j=len - 1; j>i; j--)
    {
        var compareId = visibleSurfaces[j].object.id;

        /*
         * We will not compare this surfaces with any back-face
         * surfaces and those from the same object
         */
        if( thisObjId != compareId &&
            visibleSurfaces[j].visibility == visible)
        {
            for(var k=0; k<vertices.length; k++)
            {
                if(isPointInsidePolygon(visibleSurfaces[j],
                    vertices[k]))
                    vertices[k].hidden = compareId;
            }
        }
    }

    /*
     * This is to check if all vertices of this surface is hidden by
     * any surfaces of the same object. If so, this surface can be
     * removed.
     */
    visibleSurfaces[i].checkVisibility();
}
var updatedVisible = new Array();
var count = 0;
var next = invisibleSurfaces.length;

for(var i=0; i<visibleSurfaces.length; i++)
{
    if(visibleSurfaces[i].visibility == visible)
    {
        visibleSurfaces[i].setVisibility(visible);
    }
}

```

```

        updatedVisible[count++] = visibleSurfaces[i];
    }
    else
        invisibleSurfaces[next++] = visibleSurfaces[i];
    }
    return new Array(updatedVisible, invisibleSurfaces);
}

/*
 * This function is to determine if a point is located inside a polygon
 * surface.
 * @para polygon is a polygon we want to check.
 * @para point is a point which we want to find if it is inside a
 * polygon.
 * @return 1 is the point is located inside the polygon surface;
 * otherwise return 0.
 */
function isPointInsidePolygon(polygon, point)
{
    var vertices = polygon.vertices;
    var E = new Array();
    var Q = new Array();
    var len = vertices.length;

    /*
     * Create a vector between 2 vertices of the polygon and a vector
     * between a point and each vertex.
     */
    for(var i=0; i<len; i++)
    {
        if(i == len - 1)
        {
            var diffX = vertices[0].xp - vertices[i].xp;
            var diffY = vertices[0].yp - vertices[i].yp;

            E[i] = new Vector(diffX, diffY, 0);
        }
        else
        {
            var diffX = vertices[i+1].xp - vertices[i].xp;
            var diffY = vertices[i+1].yp - vertices[i].yp;
            E[i] = new Vector(diffX, diffY, 0);
        }

        var diffX = point.xp - vertices[i].xp;
        var diffY = point.yp - vertices[i].yp;

        Q[i] = new Vector(diffX, diffY);
    }

    var k = 0;
    /* result is 1 if the point is inside the polygon; otherwise 0. */

```

```

var result = 1;
while(k < len)
{
    var crossProduct = 0;

    if(k == len - 1)
        crossProduct = E[k].crossVector(Q[0]);
    else
        crossProduct = E[k].crossVector(Q[k+1]);

    if(crossProduct.C < 0)
    {
        k = len;
        result = 0;
    }
    else
        k++;
}
return result;
}

/*
* This method is to check if this surface is occluded by any other
* surfaces of different objects.
*/
Polygon.prototype.checkVisibility = function()
{
    var vertices = this.vertices;
    var type = this.object.shape;
    var id = this.id;
    var len = vertices.length;

    if(len == 3)
    {
        if(vertices[0].hidden >= 0 &&
            vertices[0].hidden == vertices[1].hidden &&
            vertices[0].hidden == vertices[2].hidden)
            this.setVisibility(invisible);
        else
            this.setVisibility(visible);
    }
    else if(len == 4)
    {
        if(vertices[0].hidden >= 0 &&
            vertices[0].hidden == vertices[1].hidden &&
            vertices[0].hidden == vertices[2].hidden &&
            vertices[0].hidden == vertices[3].hidden)
            this.setVisibility(invisible);
        else
            this.setVisibility(visible);
    }
    else if(len == 8)
    {
        if(vertices[0].hidden >= 0 &&

```

```

        vertices[0].hidden == vertices[1].hidden &&
        vertices[0].hidden == vertices[2].hidden &&
        vertices[0].hidden == vertices[3].hidden &&
        vertices[0].hidden == vertices[4].hidden &&
        vertices[0].hidden == vertices[5].hidden &&
        vertices[0].hidden == vertices[6].hidden &&
        vertices[0].hidden == vertices[7].hidden)
        this.setVisibility(invisible);
    else
        this.setVisibility(visible);
    }
}

/*
 * This method is to assign "seen" as default hidden property value to
 * every polygon vertex.
 */
Polygon.prototype.resetHiddenProperty = function()
{
    var vertices = this.vertices;

    for(var i=0; i<vertices.length; i++)
        vertices[i].hidden = seen;
}

function drawObjects()
{
    var count = 0;

    /* Create a transformation matrix of a current perspective. */
    perspective.translate(g_tx, g_ty, g_tz);
    perspective.rotate((g_rx % 360)*0.017453293,
        (g_ry % 360)*0.017453293, (g_rz % 360)*0.017453293);

    /*
     * Get viewing coordinates and projections of this current
     * perspective.
     */
    var obj_len = g_objects.length;

    for(var i=0; i<obj_len; i++)
        g_objects[i].getVCandProjection();

    /*
     * Arrange the order of drawing objects according to the depth of
     * their center.
     */
    quicksortCenter(g_objects, 0, obj_len - 1);

    var countVisible = 0;
    var countInvisible = 0;
    var visiblePolygons = new Array();
    var invisiblePolygons = new Array();

```

```

for(var i=0; i<obj_len; i++)
{
    var vertices = new Array(4);
    var a_polygon = new Array();
    for(var k=0; k<g_objects[i].num_faces; k++)
    {
        vertices = g_objects[i].face[k];
        a_polygon[k] = new Polygon(vertices, k, g_objects[i],
            g_objects[i].plane[k]);
    }

    var a_polygon_len = a_polygon.length;

    /* Arrange the rendering order according to their depth.*/
    quicksortDepth(a_polygon, 0, a_polygon_len - 1);

    /*
    * Separate polygons into two groups, visible and invisible
    * polygons.
    */
    for(var k=0; k<a_polygon_len; k++)
    {
        /* Check if this polygon is visible. */
        if(a_polygon[k].isVisible())
        {
            a_polygon[k].setVisibility(visible);
            visiblePolygons[countVisible++] = a_polygon[k];
        }
        else
            invisiblePolygons[countInvisible++] = a_polygon[k];
    }

    /* Check hidden surfaces. */
    var results = removeHiddenSurface(visiblePolygons,
        invisiblePolygons);

    visiblePolygons = results[0];
    invisiblePolygons = results[1];

    /*
    * To optimize the calculation time, we calculate shades only for
    * the unhidden front-face polygons.
    */
    for(var i=0; i<obj_len; i++)
        objects[i].getShade();

    var invi_len = invisiblePolygons.length;
    var vi_len = afterVi;

    for(var i=0; i<invi_len; i++)
        polygonObj[i] = invisiblePolygons[i];

    for(var i=0; i<vi_len; i++)

```

```

        polygonObj[i + invi_len] = visiblePolygons[i];
    }
    for(var i=0; i<invi_len; i++)
    {
        /* Clear data in svg polygon tags. */
        polygonSVGTag[i].setAttribute("points", "0,0");
        polygonSVGTag[i].setAttribute("fill", "none");
        polygonSVGTag[i].setAttribute("stroke", "none");
    }

    for(var i=invi_len; i<polygonSVGTag.length; i++)
    {
        /* Clear data in global svg polygon tags. */
        polygonSVGTag[i].setAttribute("points", "0,0");
        polygonSVGTag[i].setAttribute("fill", "none");
        polygonSVGTag[i].setAttribute("stroke", "none");

        var projection = polygonObj[i].vertices;
        var points = polygonObj[i].stringOfPoints;
        polygonSVGTag[i].setAttribute("points", points);

        var url = "url(#" + polygonObj[i].object.id +
            polygonObj[i].id + ")";
        polygonSVGTag[i].setAttribute("fill", url);
    }

    /* Reset the perspective matrix to be an identity matrix. */
    perspective.reset();
}

```