Navigating/Browsing In 3D with WebGL

Xinpeng Yang
June 2010

Bachelor Thesis, 15 hp, C
Computer Science

Computer Science Program
Examiner: Stefan Seipel
Co-examiner: Fredrik Bokman
Supervisor: Stefan Seipel
Navigating/Browsing In 3D with Web GL

by

Xinping Yang

Institutionen för matematik, natur- och datavetenskap
Högskolan i Gävle

S-801 76 Gävle, Sweden

Email: nfk08xyg@student.hig.se

Abstract

The use of the internet has become very important in our daily life. At present, web pages are mostly shown as two-dimensional content. Three-dimensional web pages are rarely seen on the Internet. In this thesis, the main task is to develop an interactive 3D (three-dimensional) web prototype for the company Interactive TV Arena. This three-dimensional web prototype will be integrated to the OSIA (open server interface architecture) project which is a project in Interactive TV Arena. In order to implement the prototype, three main tasks were addressed: The first is creating the framework of the prototype with XML files. Secondly, a parser framework was built with JavaScript classes. The third task is to render the prototype with WebGL in real time. The prototype can be used to develop the 3D web pages in modern browsers. This thesis has completed most of the aforementioned tasks and the result of it can be used to develop some simple 3D web pages.

Keywords: OSIA, WebGL, XML, 3D
Contents

1 Introduction .................................................................................................................. 1
  1.1 Aim ......................................................................................................................... 1
  1.2 Methodology .......................................................................................................... 1

2 Backgrounds ............................................................................................................... 2
  2.1 Interactive TV Arena ............................................................................................. 2
  2.2 OSIA Project .......................................................................................................... 2
    2.2.1 What is OSIA? ................................................................................................. 2
    2.2.2 What is the OSIA Platform? ......................................................................... 2
  2.3 WebGL .................................................................................................................. 3
  2.4 Design Pattern Description .................................................................................... 3

3 Implementation ........................................................................................................... 5
  3.1 WebGL rendering environment .............................................................................. 5
  3.2 Implantation of Architecture design ...................................................................... 5
    3.2.1 Model layer ...................................................................................................... 5
    3.2.2 View layer ....................................................................................................... 6
    3.2.3 Control layer ................................................................................................... 7
  3.3 The structure parser ............................................................................................... 8
    3.3.1 Parser for the control layer structure ............................................................. 8
    3.3.2 Parser for the model layer structure .............................................................. 10
    3.3.3 Parser for the view layer .............................................................................. 11
  3.4 Implementation of rendering ................................................................................ 12
    3.4.1 WebGL rendering pipeline .......................................................................... 13
    3.4.2 Render processing ......................................................................................... 13
    3.4.3 Basic Matrix mathematics ............................................................................ 14
    3.4.4 Shader loading ............................................................................................... 14
    3.4.5 Scene rendering ............................................................................................. 16
  3.5 Integrate the project into OSIA project ................................................................. 25

4 Discussion ................................................................................................................... 26

5 Conclusion ................................................................................................................... 28

6 Acknowledgements .................................................................................................... 29

Abbreviations ................................................................................................................. 30

Reference ....................................................................................................................... 31

Appendix A: site.xml ..................................................................................................... 32
Appendix B: document.js ............................................................................................ 34
Appendix C: process of parser xml files ................................................................. 36
1 Introduction

Interactive TV Arena focuses on interactive services in the TV media. Their interactive TV services are used on mobile, broadband, and the DVBT (Digital Video Broadcasting Terrestrial). But now the user interface mostly is displayed as two-dimensional WebPages. In order to create more interesting services, a three-dimensional user interface for the web is needed. So the goal of this thesis is to research and evaluate solutions, and then develop a prototype 3D web browser environment in modern browsers. This environment should be able to process data files to display information in 3D, in a way similar to how browsers display normal web pages. In addition it should also be able to navigate this information in an intuitive and simple way.

1.1 Aim

The aim of this paper is to develop a prototype 3D web browser environment in modern browsers. It should be possible to use for the three dimensional user interface development of the set top box in the future.

The tasks to be carried out in this work are listed below:

1. To implement the basic framework for rendering. The framework is created with XML files.
2. To parse the XML framework files with JavaScript classes.
3. To implement the basic elements such as object, texture, text, light, shading, animation, and camera. And the elements can be rendered in the prototype.

1.2 Methodology

The project is based on programs which were primarily developed at Interactive TV Arena. The work includes these methods:

- For getting familiar with the WebGL JavaScript API, the JavaScript programming language and XML. Literature studies were done with the help of Internet and books.
- For further definition of the requirements for Interactive TV Arena and to get familiar with management processes. Information was gathered by:
  - Reading internal project documents.
  - Meetings, discussions, mail dialogues during the day-to-day work at Interactive TV Arena.
  - Learning some knowledge from the interactive user interface design books and learn the primary OSIA project in the company.
2 Background

This chapter is aimed to give the reader a better understanding of essential concepts and the background of this work. This chapter also gives a short description of Interactive TV Arena and the services of the company, as well as the OSIA project. Finally a basic architecture description.

2.1 Interactive TV Arena

The project was done in Interactive TV Arena which is located in Gävle, Sweden. The company deals with the services for TV Media. This developing institution started working with interaction television in the year of 2000. This is a small company with 8 regular employees. The company is dealing with different services like 1) ITV Lab, the new ideas and services are tested here by companies/organizations. 2) ITV Screening, here the testing and evaluation is being done by the panel of almost 1000 households. 3) ITV Format Panel, a number of production companies and broadcasters can access the company’s infrastructure and/or ITV panel to test the new TV formats. 4) ITV Research, it includes Labs, tests panels, networks and infrastructure for the research groups. 5) ITV Cluster, It is the combination of networks and skills. Partnership is also offered to those who want to contribute their resources and expertise. This thesis is a kind of ITV Research service.

The services of the company are focused on the media service in different devices, like computers, mobile phones and DVBT. It provides the services between the service providers and consumers, and the services are mostly focused on the media content. They also provide the service with set top box, which can show the TV service with interaction with the user interface of the set top boxes. Beside the media service they also conduct a research project (OSIA which we have referred above) relative to interactive television.

2.2 OSIA Project

2.2.1 What is OSIA?

There are two broader aspects of OSIA, one is the research and the other is the technical aspect. OSIA is the abbreviation of Open Service Interface Architecture [1]. It is an open generic framework for systems and services based on web technology, now mostly focused on the media content.

The other is the technical aspect. The researcher aims to find some new ways or technical approaches to make communication more efficient and useful between customers and providers. The technical aspect mostly focuses on the implementation of the OSIA software. It is about realization of the research results. It can be described in a simpler way, as a library for any developer who wants to provide the media services basic web technologies to customer. For example, some developers can add some code to the software to enhance the functions as they want, and also make some changes of the code, they can also put some new media contents service for their customers.

2.2.2 What is the OSIA Platform?

As described above, the OSIA platform is the software which is the implementation of the OSIA framework. It can provide the service based on web technologies, but now
mostly it is focused on the service of provide or handling media content. Figure1 describes the OSIA actor’s relationship.

![Figure 1. Example of OSIA actor’s relationship](image)

From Figure 1, we can see that the service provider can download and install the OSIA software. After installation of the OSIA software, the management system will be in the OSIA platform, and the processing is clear to describe as Figure 1 shows. One actor is the service providers, they produce the content like the media content, and immediately it can be available to selected groups or public. The other actors in the OSIA platform are not providing any content by their own, they only act as the broker of the service providers. They provide the “portal”, which can get the services from the service providers. And the third actors are the users. The user can enjoy the service from the portal which has been created by the brokers. So it creates the relationship between the providers and the brokers. It is very convenient that the portals can be easily aggregated. The providers can provide the services to one or more brokers. The OSIA platform provides a convenient architecture, because it based on a Lego-like design.

### 2.3 WebGL

WebGL is the abbreviation of Web Graphics Library; it is a low-level three dimensional graphic JavaScript API for developing three dimensional web applications. It is based on OpenGL ES 2.0, with the character of cross-platform and open. One of the advantages is that WebGL can be used to build in 3D graphics into the webpage without any external plug-in.

WebGL is a new technology and its specification was released on December 10 2009. It can be found in Khronos [2]. It supports GLSL (OpenGL Shading Language) shadings. It can access GPU (Graphic Processing Unit) with shading programming to deal with 3D graphic development.

At present, only the latest browsers support WebGL. The latest browsers are: Google Chrome, Minefield (latest but not final version of Firefox), Apple (Safari version 4.0).

### 2.4 Design Pattern Description

Three layers make up the implementation of this prototype which is split into model/data, view and control. This follows the design pattern “Model-View-Controller” (MVC). The layers are described as follows:
• **Model Layer**

The model or data layer consists of XML markup languages that define the information and visualization of the information. These are separated into shape XML and material XML.

Shape XML defines objects like spheres, cubes, cones etc, using primitives in WebGL such as lines, dots, polygons to build.

Material XML defines the attributes of the texture which will be mapped to the object.

• **View Layer**

In order to create a view, a number of JavaScript parsers will need to be constructed to combine and render the model (data files). The XML files defined in the view part are light, text, and visual style XML files.

Light XML defines the light type, position, direction.

Text XML defines the text which can be mapped to the object or set in the scene. It includes the size, position, and text content.

Visual style XML includes the color of the object, shadings, and also some basic animation, like translation, rotation and scale.

• **Control Layer**

The control layer is responsible for processing events and alters the view in order to allow interaction with the site. The XML file defined in the control layer is site.xml. The site.xml can import the model and view layer content, like the light, text, shape, and aggregate all the XML files into the site.xml file. So it can control the whole scene which the developer needed. The developer only controls the site.xml. He/she can build the relative XML files, such as the shape, camera, light, then render the site.xml and get a three dimensional scene in the webpage.
3 Implementation

The implementation contains five sections. 1) Simply describing the WebGL rendering environment, how to use the latest browser to browse the three dimensional webpage. 2) The architecture design with XML files 3) Parsing the structures with JavaScript classes. 4) Rendering it in real time with WebGL. The implementations of the basic elements are camera, texture, text rendering, shading loading, light, shape, JSON (JavaScript Object Notation) model. 5) Integrate the project into OSIA project.

3.1 WebGL rendering environment

Unfortunately not all browsers support WebGL. Because WebGL is a new technology and it is still in developing phase, only the latest browsers support it. The browsers which can support it are Firefox, Chrome, and Safari. This project has been tested and debugged in the Firefox and Chrome browsers. Before it starts the WebGL three dimensional pages, it requires changing some attributes setting on the browsers. Here we only describe the process about attribute settings in Firefox and Chrome in Windows platform.

To start Firefox, it requires the _nightly builds page_[3] to get the proper version for the computer. It needs proper installation to start Minefield (latest but not final version of Firefox), then requires to go on the “about: config” page. It will appear on searching of the “webgl”, and then set the value of “webgl.enabled_for_all_sites” variable to “true”. And it also needs to set the value of “dom.ipc.plugins.enabled” to “false”.

For Chrome, it requires to go to the _continuous integration page_[4] and download chrome-win32.zip. Then unzip the file, install it and set the command in the attribute of the shortcut/icon item. The last step is to go to properties and add the “--enable-webgl --disable-sandbox “command in the back of target dialoged box.
For more detail to set the browsers in different operating system platform, the WebGL lesson webpage [5] is useful.

3.2 Implementation of Architecture design

3.2.1 Model layer

In the model layer, two XML files should be built. One is shape.xml, another is material.xml.

The shape.xml is building the basic shape of the object. In WebGL, it has a drawElement (*) or drawArray (*) function to draw the object. To draw the shape, the developers need to include different contexts: the vertices, normal, texture coordinates, and indices. The XML is as follows:

```
<shape id="shape id " type="TRIANGLES">
   <vertices >vertices coordinate</vertices>
   <normals>vertices normals</normals>
   <texCoords>texture coordinates</texCoords>
   <indices >index of the vertices</indices>
</shape>
```
The type attribute used in the shape is the corresponding WebGL draw types. It has many types to draw the object, for example, TRIANGLES means to draw the object with triangles. The context of the <vertices> tag is vertices coordinate, <normal> is the normal of each vertex, <texCoords> is the texture coordinates and <indices> is the index of the vertices.

The material.xml stores the texture data and some color of the surface in the texture. It includes the source path of the texture and the texture coordinates. The structure of the file is as follows:

```xml
<material id="treematerial" specular="0" color="#FFFFFF" transparence="0.5">
    <texture id="treetexture" src="url" />
</material>
```

The developers can define the identity of the material, and the specular attributes for the specular light which affects the texture. It also can set the transparence of the texture, 0 means no transparence, 1 means the totally transparence of the texture. The color used in hexadecimial means to use the numbers base on 16. For example, 1 to 9 is same number in decimal number, but 10 to 16 using A to F instead. This format is more familiar with the developers who have the experience of developing webpage.

### 3.2.2 View layer

The view layer includes the light, text, visual style, camera and animation XML files. The animation XML defines the animation of the object in the scene. Above five XML files make up the view layer.

The light XML file defines the light in the scene. It includes the position, color, and direction. The position is about where to put the light, color is the light color, and the direction defines the direction of the light shine. The identification defines the type of the light. The developer can define it as spotlight, direction light etc. The light XML is designed as follows:

```xml
<light id="#spotlight">
    <position>1.0,1.0,1.0</position>
    <color>#FFFFFF</color>
    <direction>1.0,1.0,1.0</direction>
</light>
```

This light XML defines a spotlight light, the position of the light is in the position [1.0, 1.0, 1.0], these three coordinate is standard of x, y, z axis in the scene. The color is using hexadecimal which is the same as mentioned above.

The text XML file is a design file which stores the text attribute and value. In the text XML, it should include the font, the size of the text, and also it can include the position, color, rotation, and the text content. The text XML file is designed as follows:

```xml
<text id="#text1" font="font type" size="text size">
    <position>1.0,0</position>
    <color>#FFFFFF</color>
    <rotation>1.0,1.0,1.0</rotation>
    <textArea>input the text content</textArea>
</text>
```
The text identification is ‘#text1’, position, rotation, color definition are similar as the light XML file. The developers can put any string in the text area, and can choose proper font, and they can also set the size attribute of the text.

Visual style XML file is the design file which stores the style of the object. In this XML file, we define the shading and the color as the styles. It describes the different shadings, like Phong shading, bump mapping shading, display mapping shading etc. It also includes the basic static animations, like translation, rotation and scale. The xml file of the visual style is as follows:

```xml
<visual_style id="#visual1">
  <color>#FFFFFF</color>
  <translation>1.0,0.0,0.0</translation>
  <rotation angle=’90’>0.0,1.0,0.0</rotation>
  <scale>0.5,0.0,0.0</scale>
  <glsl type="vertex">vertex.vertex</glsl>
  <glsl type="fragment">frag.fragment</glsl>
</visual_style>
```

Camera is the visual angle in the scene. We define the camera file in the view layer. The developers can name the camera with id. The file has the basic attributes of the camera, like position and rotation. The camera XML file is as follows:

```xml
<camera id="#camera2">
  <position>1.0,1.0,1.0</position>
  <rotation>1.0,1.0,1.0</rotation>
</camera>
```

The animation.xml only allows the developers to design simple animations. Some advanced animations, like bone animation and frame animation will be improved in the future. The basic animation is wobbling the object. The time is the frequency of wobbling the object and the size attribute is the amplitude of the object from near to far. And the direction is the direction with x, y, z axis. The structure is very simple:

```xml
<animation id="#animation1" time="30" size="0.5">
  <direction>1.0,1.0,1.0</direction>
</animation>
```

### 3.2.3 Control layer

In the control layer, all the XML files in the model layer and view layer will be combined together. All of the data in the XML files are included in the scene. So the scene should include the object with its visual style, light, text and visual style of the scene. The site.xml file has an <import> tag which can import other XML files with different file paths. And it also can integrate all the xml files in the model layer like shape and material, and view layer like light, text, animation, visual style, and camera together. So the only operation of the control layer is to parse the site.xml file, and transform all of the rendering data in this xml to one JavaScript class, and finally get the whole structured data for rendering in real time.
The Scene structure should include some scene attributes, like the background color, camera, and the fog. Inside the scene, it should have the light, object, text and visual style.

For the object, it should have shape and material which comes from the model layer. It also has some animation and visual style which comes from the view layer. The structure of the scene is as below:

```
<scene id="#scene1" camera="#camera1" ambient_color="#FFFFFF" fog_type="fog">
    <object id="#object1" shape="#triangle" animate="#animation1"
            material="#wallmaterial" visual_style="#visual0" />
    <object id="#object2" shape="#triangle" animate="#animation1"
            material="#wallmaterial" visual_style="#visual0" />
    <light id="#light1" light="#mainlight" />
    <light id="#light2" light="#mainlight" />
    <text id="#text1" text="#text1" />
    <text id="#text2" text="#text1" />
    <visual_style id="#visual1" visual_style="#visual1" />
    <visual_style id="#visual2" visual_style="#visual1" />
</scene>
```

3.3 The structure parser

3.3.1 Parser for the control layer structure

From the above, we have built the whole structure of the model layer, view layer, and control layer. So to parse the structures, we only need to parse the site.xml file, and get the whole dataset in the xml files. For this part, we have to learn some XML knowledge, a useful source of XML knowledge can be found at w3schools.com[6]. The basic idea is to parse the XML elements to get the data in the XML file. At first get the root tag, and then work from top to bottom to parse the whole XML document tree. For a simple example, we can use the simple code as follows:

```javascript
xhr = new XMLHttpRequest ();
xhr.open ("GET", url, false);
xhr.send ("");
//Get the XMLDocument (DOM) of the XMLHttpRequest
var xmlDoc = xhr.responseXML;
var root = xmlDoc.documentElement.childNodes;
```

The XMLHttpRequest () function sends the XML http request, then use the open ("GET", url, false) function (the url is standard of the path of the file) to open the xml file with the url path. Send ("") function will finish the requirement. Then we can get the content with xhr.responseXML. The responseXML is the content of the xml file. Then we get the root element list with xmlDoc.documentElement.childNodes. From the root element, we can parse the xml trees, with the parser functions.

From above structure, the basic idea is parser the site.xml file. The site.xml includes all the structures which we designed in the related xml files. In order to describe it more clearly, we give an example of a control xml file which includes the whole structure, we call it site.xml. Look at appendix A site.xml.
The simple figure as follow:

Figure 2 Structure of site.xml

In the OSIA project, the portlet was built by different XML files. And it also has a combined XML file to control the whole XML files together. The <import> tag can realize this task, it only imports the path of the different xml files, like shape.xml, material.xml, text.xml, visual_style.xml, animation.xml, camera.xml, light.xml and scene.xml. And all of the XML files can be located in different paths, so the developer can focus on the different part. This kind of framework will be very clear and easy for the developer to build the 3D webpage developing. The basic structure is built as above, the next step is how to parse the xml file, and get the data into the respective JavaScript classes. The structure we can describe in Figure 3 as follows:
There are many different tags in the site.xml, such as <import>, <shape>, <material>, <light>, <text>, <visual_style>, <animation>, <camera>, <scene>. For each tag, it also has the sub tags inside these tags. In order to parse the XML file, it is necessary to build relative JavaScript classes to these tags. Then it makes a parser function in each relevant class. For our design, we build the relevant classes, such as shape, material, light, text, visual_style, animation, camera, scene classes. Each class has its own id, attribute, parser function.

To parse the site.xml file, we write the document.js. The document.js file can be found at Appendix B: document.js.

From the code, we can see the Document class, it has a load(url) function. The url is the path of the site.xml file. The load(url) function will load the whole tags in the site.xml beside the import tag. And it also has a loadXML(url) function, where the url is the path for those xml files which need to be imported, like shape.xml, light.xml and so on. So parsing the <import> tag can be realized in the loadXML (url) function. For other sub tags, we create the relevant JavaScript classes, and then use the parser functions in the relative classes.

### 3.3.2 Parser for the model layer structure

Shape class:
The shape class has the id, type attribute, and also has five sub tags: <vertices>, <normal>, <texCoords>, <indices>, <color>. Figure 4 shows the process of parser shape.xml.
The shape has id, type attributes. The parser function will parse the `shape.xml`, it gets the shape tag attributes like id and type data, and set the value to the attribute id and type in the shape JavaScript class. Inside the shape class, it has five sub classes: they are `vertices`, `normal`, `texCoords`, `indices` and `color`. Some of these classes also have attributes or values. The parser of the respective sub tags gets the value of the attribute or text content in the XML file, then sets it into the relative sub classes which are included in the shape class.

Material class:
In the `material.xml`, there are two tags, `<material>` and `<texture>`. The `<material>` tag has id, specular, color, transparence attributes, and `<texture>` also has the attributes of the texture. The parser function will also parse the `material.xml`, and set the value of the relative attributes to the material class. When parsing the tag `<texture>`, the `<texture>` value is set to the texture class. Figure 5 shows the process.

### 3.3.3 Parser for the view layer

In the view layer, the object, text, visual styles will be included in the scene. Here we only describe the process of parsing the scene class. The other classes like light, text,
animation, camera, visual style classes can be seen in the figures in the Appendix C. The parser way of these classes are similar to shape and material classes.

Figure 6 Process of scene.xml

Figure 6 shows the parser processing of scene.xml. There are four sub tags in the scene <object/> , <light/> , <text/> and <visual_style/>. In order to parse the scene, four relative classes should be created. They are object, light, text, visual style classes. Each class has a parser function. The parser function in the site.xml will iterate the XML tree. When a <scene> tag is found in the tree, it should create a new scene class, and also parse the scene with its parser function. Then it will parse the object, light, text, and visual style in the scene. In the object, it has four attributes, shape, animation, material and visual style. The object has a parser function too. It will parse the attributes in the <object> tag, then read the relative data, and set them as attributes or values in the relative classes. The same method is used to parser light, text, and visual_style.

3.4 Implementation of rendering

The basic idea of rendering is to give a simple interface to the developers, all rendering detail should be encapsulated in the namespace namo. For our design, rendering a 3D scene will be very easy, needs only two lines code.

namo.load("site.xml");
namo.webGLStart();

The first line, it is the load(url) function, which loads the site.xml. The namespace namo has a Document attribute, after load(url) function was finished, all the data which stored in the xml files should be transferred to the Document JavaScript class. Then we start the second function namo.webGLStart(), it will render the 3D scene with WebGL. The rendered data all come from the XML files. In this section we will describe the process of rendering the XML file structure with WebGL. There are four phases in this section,
WebGL rendering pipeline and processing, basic matrix mathematic building, shader loading and scene rendering.

### 3.4.1 WebGL rendering pipeline

WebGL support OpenGL ES 2.0, so the rendering pipeline is similar to OpenGL shader rendering pipeline. In WebGL, the HTML5 canvas element and its memory management is used with JavaScript language [7]. Figure 7 shows the rendering pipeline with WebGL.

![WebGL rendering pipeline](image)

**Figure 7. WebGL rendering pipeline**

As Figure 7 shows, the flow of the data transform is simple, and it also shows that the GPU deal with the data flow with the buffers. In the high level, vertex shader can transform the variable like attribute and uniform, but WebGL processes the data with buffers. In WebGL, there is a function drawArrays. It can pass the variable along to the vertex shader.

Between vertex shader and fragment shader, the data is transferred with varying variables. The vertex shader deals with the vertex, like the gl_Position which contains the co-ordinates of the vertex. Hence we can use some mathematic operating on this vertex, such as model-view matrix, projection matrix, normalize matrix. The fragment shader deals with the color of those interpolated points. It has the gl_FragColor which returns the color of texture or geometry object. So the color operation can be controlled by the fragment shader. The Illumination algorithm like Phong shading can be calculated in the fragment to interpolate the color Value. For more details for learning the WebGL rendering pipeline, the WebGL learning tutorial [8] is helpful.

### 3.4.2 Render processing

In html5, there is a canvas element, WebGL can get the OpenGL ES 2.0 context by canvas.getContext("experimental-webgl") function, so we can do like this:

```javascript
var gl = canvas.getContext("experimental-webgl");
```

The variable gl can get the whole context of OpenGL ES 2.0. We can make some JavaScript code to use the context of OpenGL graphic API flexibly. Now we can build WebGL start(*) function to rendering scene. The entire 3D scene will be rendered in the canvas. Six steps combine the processing of rendering:

1. Get the canvas element.
2. Initialize the gl variable to get the OpenGL context.
3. Initialize the vertex and fragment shaders. Compile and link the shader, and load it to the WebGL program. Then set the variables which come from the shaders, like **uniform**, **attributes**.

4. Initialize the buffers. When building the object, there is some coordination of the points, texture, and color. This data should be put into the WebGL buffers, as Figure 7 shows. Buffer should be created which stores the data. This stored data is needed for rendering.

5. Draw the scene.

6. Define a WebGLstart (*) function to start rendering in the html file.

Follow the six steps, we make some simple code to describe the process of rendering.

```javascript
var canvas = document.getElementById("canvas id");
initGL (canvas);
initShaders ();
initBuffers ();
gl.clearColor (0.0, 0.0, 0.0, 1.0);
gl.clearDepth (1.0);
gl.enable (gl.DEPTH_TEST);
gl.depthFunc (gl.LEQUAL);
drawScene ();
```

**3.4.3 Basic Matrix mathematics**

In the WebGL, there are no basic Matrix functions, so the basic matrix mathematic operation should be implemented by us. When we render the scene, there are many operations needs matrix operations. To realize this mathematic operation is very difficult, but luckily, we can get some open source. In our project, we used the open source code with name sylvester.js. It can be downloaded from Sylvester [9].

By using the sylvester.js, we write a JavaScript class **matrix** to control the whole matrix operating in the rendering. The matrix operations are **mvPushMatrix()**, **mvPopMatrix()**, **loadIdentity()**, **multMatrix(***)**, **mvTranslate(***)**, **mvScale(***)**, **mvRotate(***)**. These operations can move the object which we need to render. For the view matrix, we build the **makeLookAt(***)**, **makeOrtho(***)**, **makePerspective(***)**, **makeFrustum(***)**, **makeOrtho()**. They are similar to the functions **gluLookAt(***)**, **glOrtho(***)**, **gluPerspective(***)**, **glFrustum(***)** in GLUT[10] (the opengl utility toolkits). To understand the implementation of this matrix mathematic operation, some graphic books or some basic linear algebra knowledge are useful.

**3.4.4 Shader loading**

Shaders are also built with XML files in this project. The visual_style.xml has a tag `<glsl>`. This tag only provides the path where the shader is located. The XML file contains the shaders’ content and the variables with `<SCRIPT>` and `<variable>` tags. Parser the shader XML files to get the shader content is made the same way as we parse the XML which we mentioned before. The difficulty is how to handle the variables in the shaders. In order to handle the variables, we set the variables as tag in the xml, below is an example of the vertex shader file:
In the xml file, we set two different tags, one is <SCRIPT>, and another is <variable>. The <SCRIPT> tag includes the context of the shader, id, and type attributes. The <variable> also set the id and type attributes, the id can be uniform, attribute or varying, the type is type of matrix or vectors, for example. Then we build a glsl JavaScript class, we can parse the shader xml file and store the data in the relevant class.

When parsing the xml files, two things need to be done. One is store variables, the other one is using WebGL program to deal with the context of shader. The glsl class has a parser() function which can parse the shader XML file, and also get the relative data in shader XML files, then set them as the attributes of glsl class. The glsl has three attributes: uniform, varying, and attribute, they are array variables. The parser function in glsl class, parses the shader XML files, and puts the XML data to relevant variables in glsl class. The Figure8 can be easy to understand the relationship between the glsl class and shader xml file.

Figure 8  Relationship between shader XML and glsl class
3.4.5  Scene rendering

Scene rendering renders the scene in 3D environment with WebGL, the rendering dataset comes from the XML files. It renders all of the elements in the site.xml file. In the scene, there are four rendering elements, object, text, light, and visual style. The input control, like camera, keyboard, and mouse are also included. This section contains the rendering structure of scene, camera build, object rendering, text rendering, light rendering and visual style rendering.

3.4.5.1  Structure of scene rendering

Figure 9 is the structure of scene rendering. There are object, text, style, light four arrays, and a camera class in the scene class. As the figure shows, the rendering is started in the scene.draw() function. In the scene.draw() function, it initialize the light, style, text, object at first, then draws the object with the object.draw() function. The object.set() function sets style, text, animation, material and shape of the object. When draws the object, it also sets the style, text, animation, material at first, then draw the shape. The basic rendering process is clear now, first draw the scene with scene.draw(), then draw the object with object.draw() function, the last step is to draw the shape or JOSN object with the draw() function in the shape or JOSN class.

3.4.5.2  Camera building

Building the camera in WebGL is made in the same way as in OpenGL. But in WebGL, it does not have matrix mathematic function, so the matrix operation should be built. The matrix operation was mentioned before. The camera needs only three steps to build. The first step is build the projection matrix. When we call this matrix, only the projection transformation will be called, and transform the eye space coordinates into clip coordinates. The second step is building the model view matrix, this matrix should transform object space coordinates into eye space coordinates. So the camera only transforms the model view matrix. The last step is to build the lookat(eye.x, eye.y, eye.z, center.x, center.y, center.z, up.x, up.y, up.z) function. Set the eye as variable parameter, and then put it in the camera class. Then change the parameters, when we input the keyboard. We set the left and right cursor keyboard to change the value of eye.x, up and down cursor keyboard to change the value of eye.z, and the pageup or pagedown to control the eye.y. The object position is the value of center.x, center.y, center.z.
3.4.5.3 Object rendering

Object has four elements: shape, material, animation, and visual style. Shape is the geometry which is built by the developers. Material has the texture element, and the texture also has some attributes such as transparent, color. For animation we just define the wobble animation which wobbles object with x, y or z axis. We define 1 as the biggest size of wobble and 0 as the smallest size. Visual style is the style of the object, it includes the color of the object, rotation, scale, translation, and shader. In the following sections we will describe how to render it one by one.

3.4.5.3.1 Shape rendering

The shape rendering, there are two difficulties to deal with. One is how to get the data from the site.xml file, and then put this data into the buffers of WebGL. In the code, we create buffers with `initbuffers()` function. The shape has five tags `<color>`, `<indice>`, `<normals>`, `<vertices>`, and `<texCoords>`, so it should create five buffers. If the developer just defines part of these tags, then the problem will be how to track the buffers automatically. Another is how to map the animation and style to the geometry.

To solve the first problem, we usually set an attribute array, the index of buffers in the attribute array is the name of the buffer. For example, if you like to store the vertices buffer, we can define the buffer like this:

```javascript
attribute ['vertices']= gl.createBuffer();
```

When we need to use it, we can track the buffers automatically, see the following codes:

```javascript
var x=null;
for(x in attribute)
    using the attribute[x] buffer
```

To get the data from shape.xml, only check the document which has stored all of the data in the XML structure files. The code below shows how to get the data from shape.xml, and then put it into the buffer.

```javascript
var x=null;
for(x in namo.glsl.attribute)
{
    if(namo.glsl.attribute[x].name=='vertex')
    {
        this.createBuffers('vertex', this.vertices,3);
        if(this.indices!=undefined)
            this.createIndexBuffers('vertex', this.indices,1);
    }
    if (namo.glsl.attribute[x].name=='color')
        this.createBuffers (' color ', this.color, 4);
    if(namo.glsl.attribute[x].name =='normals')
        this.createBuffers ('normals', this.normals, 3);
    if(namo.glsl.attribute[x].name=='texCoords')
```
this.createBuffers ('texCoords', this.texCoords, 2);
}

The `createBuffers (*)` function will create the buffers in the attribute array. The first parameter is the name of the buffer, the second parameter is an array which we can get from the document. The data in the array come from the shape.xml file. The third parameter is the dimension of the array. Then it uses `gl.bindBuffer(gl.ARRAY_BUFFER, buffer)` to bind the buffer, and the `gl.bufferData(gl.ARRAY_BUFFER, new WebGLFloatArray(shape xml data ),gl.STATIC_DRAW)` function to put the XML data to the buffer.

To solve the second problem, we should describe simply how WebGL draws the geometry.

In WebGL, there are two different ways to draw a shape. One way is to draw the geometry by the function `drawArrays(type, 0, buffer.numItems)`. There are many different types in WebGL to draw the geometry. Like the `gl.TRIANGLES`, it means that draw the geometry with triangles. For detail to know the function, refer to the WebGL Draft Specification [1] which was released on December 10 2009. This function will draw the geometry by the order of buffer items. The other way is draw the geometry by the index of the buffer items. The `drawElement (type, buffer.numItems, gl.UNSIGNED_SHORT, 0)` function will draw the geometry by the index of buffer.

There are two different ways defined to draw the object, so we defined the `draw ()` and `drawElement ()` functions to draw the geometry. We are using a switch to select which type of shape to draw, see below:

drawElement (buffer)
{
switch (buffer.type)
{
case 'TRIANGLES':
    gl.drawElements (gl.TRIANGLES, buffer.numItems, namo.gl.UNSIGNED_SHORT, 0);
    break;

case 'TRIANGLE_STRIP':
    gl.drawArrays(gl.TRIANGLE_STRIP,buffer.numItems, namo.gl.UNSIGNED_SHORT, 0);
    break;
    ...
    ...
default:
    break;
}

As the code shows, the geometry can be drawn for the different types.

We draw the object with an `object.draw ()` function. Before we draw the geometry, we have set the material, style, animation in the `object.draw ()` function. The `object.draw ()` function code is below:
matrix.mvPushMatrix()
materialSet(**);
animationSet (**);
objStyleSet (**);
drawElement (**);
this.matrix.mvPopMatrix ();

So all the style and animation will be mapped to the object. For how to make a geometry with buffer in WebGL, it has a detailed description in WebGL Draft Specification [1].

3.4.5.3.2 Material rendering

Material rendering will map a texture to the object. The texture has colors and transparency attributes which can be defined by the developers. The developers can map many kinds of format images as texture to the object. There are three steps when we deal with the texture in WebGL.

The first step is to deal with the color and transparency of the image. This is done in the fragment shader. In the fragment shader, there is a gl_FragColor variable, this variable can affect the texture’s color and transparency. One simple code describes it in the following:

```html
<root>
  <variable id="varying" type="vec2">TextureCoord</variable>
  <variable id="uniform" type="sampler2D">textureUniform</variable>
  <SCRIPT id="shader-fs" type="x-shader/x-fragment">
    varying vec2 TextureCoord;
    uniform sampler2D textureUniform;
    void main (void) {
      vec4 texture = texture2D (textureUniform, vec2(TextureCoord.s, TextureCoord.t));
      gl_FragColor = vec4 (texture.rgb, alphaUniform);
    }
  </SCRIPT>
</root>
```

In this code, we set the textureUniform which will be used for rendering the color data, and the TextureCoord varying variable which can pass the coordinates from the vertex shader to the fragment shader. The function texture2D(**) can get the texture coordinates, mapping the texture RGB(red, green, blue) value to gl_FragColor, and also mapping the alphaUniform to the gl_FragColor, so the color and alpha will be controlled by the textureUniform and alphaUniform. These two uniforms can be used for rendering.

The second step is mapping the image as texture to the object. In WebGL, the function gl.createTexture () can create a texture, and also create an image with the
Image () function. Then the image is binded to the texture. The following code realizes the image as texture map to the object.

```javascript
var texture = load (url);
load=function (url) {
  var texture = namo.gl.createTexture();
  gl.bindTexture(namo.gl.TEXTURE_2D, texture);
  gl.texParameteri(namo.gl.TEXTURE_2D, namo.gl.TEXTURE_MAG_FILTER, gl.NEAREST);
  gl.texParameteri(namo.gl.TEXTURE_2D, namo.gl.TEXTURE_MIN_FILTER, gl.NEAREST);
  var image = new Image();
  image.onload = function () {
    gl.bindTexture (namo.gl.TEXTURE_2D, texture); // Bind the texture
    gl.texImage2D (namo.gl.TEXTURE_2D, 0, image, true); // Set the image to texture
  }
  image.src = url;
  return texture;
};
```

When the texture loading is finished, and then the object is drawn in the scene by using the texture and alpha uniforms which are defined in the shader. The realization can be seen in the following code:

```javascript
gl.activeTexture (gl.TEXTURE0);
gl.bindTexture (gl.TEXTURE_2D, texture);
gl.uniform1i (texture Uniform, 0);
gl.enable (gl.BLEND);
gl.uniform1i (alphaUniform,0);
```

This code will bind the texture to the object, and set the uniform to effect the color and transparence on the object. The uniform variable value will be set in the material.xml file, and the developers can define it.

The third step is how to get the data in the material.xml and use it for material rendering automatically. This step is similar to getting the data in the shape rendering. The difficulty is how to realize material rendering automatically. Like setting the attributes in the shape rendering, we can also set the uniform array with the name of the uniform. We can get the name from the uniform context in the fragment and vertex shaders. Then when we set the uniform actions, we can iterate it as following code:

```javascript
var x=null;
for(x in uniform)
  Uniform[x] =...;
```

This will iterate the uniform variables automatically. In the same way, when we bind the texture to the object, we can set it like this:
var x=null;
for(x in uniform)
   gl.uniform1i (uniform[x],0);

So the uniform variable like the color and alpha will be used automatically. For the material rendering, we define the materialSet() function in the material class, before drawing the geometry, using materialSet() function to render the texture. Now the object.draw() function code is as below:

matrix.mvPushMatrix();
materialSet(**);
drawElement (**);
matrix.mvPopMatrix();

3.4.5.3.3 Animation rendering

Animation is difficult to realize in graphic rendering. There are frame animations and bone animations. In this thesis, we just follow some requirements from the company, by realizing some basic frame animations. But it can be extended with bone animation in the future. We have followed the instructions of the company, and wobble the object along with x, y, z axis. To design the animation xml file is also easy. We only need the <time> and <animation> tags to control the animation. The time is the frequency of the wobble object, the animation defines the wobble size along the x, y, and z direction. In WebGL, the frame animation realization is to change the value of the variable between two closed frames. Then combine the translation, rotation, and scale function in the matrix class, and set some variables in these functions. The following code will describe the wobble object along with the z axis.

matrix.translation (0.0, 0.0, valueZ);
matrix.scale (scal, scal, scal);
Draw object...
Var speed=0.5;
valueZ=-10;
flag=true;
scale=0.5;
Animation = function ()
{
 timeNow= new Date().getTime();
 if (lastTime != 0) {
   var elapsed = timeNow - lastTime;
   if (valueZ<-1&&flag==true) {
     valueZ+=0.1
     scale*=1.2;
     if (valueZ>=-1)
       flag=false;
   }
   if ((valueZ>=-1&&flag==false) {
     valueZ-=0.1
     scale*=1.2;
     if(valueZ<-10)
       flag=true;
}
In the animation function, the flag will lock one of the two if sentence, so the valueZ and scale values will be change between the two close frames. When we render the object, it will be wobbling along with the z axis from -10 to -1. The frame animation is realized by changing the variable values between the close frames, by using basic matrix functions to do some animation for moving the object.

3.4.5.3.4 Object visual style rendering

Object visual style rendering means to renders the style only on the rendering object which the developer want to. The visual style has some attributes, like translation, scale, rotation, color of the object, and also shaders. In this part, we only realize the Phong shading on the object. But the structure is easy to extend in the future. For object visual style rendering, two things will be done. One is color, translation, scale, rotation styles rendering realization, and another is the realization of the shading.

Translation, scale, rotation of the object is easy to realize. It only requires reading the data from the visual style xml file. As mentioned before, after site.xml file is loaded, all the data is stored in the document class, so it is easy to find the relevant data in the object visual style xml files. The rendering code is as follows:

```
Var x=null;
For(x in namo.Document.sence ['#sence1'].object)
{
    Var translation=namo.Document.sence ['#sence1'].object[x].translation;
    Var scale = namo.Document.sence ['#sence1'].object[x].scale;
    Var rotation= namo.Document.sence ['#sence1'].object[x].rotation;
    Matrix.mvTranslate (translation);
    Matrix.mvRotate (rotation);
    Matrix.mvScale (scale);
}
```

Setting color is done in the shading, we build a Phong illumination model in the vertex shader by following this formula:

$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (L_m \cdot N) i_d + k_s (R_m \cdot V)^\alpha i_s).$$

which can be found in webpage[11] we have set the ambient, specular, diffuse color as three uniforms, then the uniform variables get the data from the visual style xml file, and map to the object. The code is written as below:

```
var x=null;
For(x in uniform)
{
    If(x == 'ambient'||x=='specular'||x=='diffuse')
        namo.gl.uniform1i (uniform[x], 0);
}
```

The second part is mapping the shader to the object. At this moment we use the simple way, only get the path of the shaders, then read the data in the shader xml file, and finally load the shader.
All objects of visual style rendering can be encapsulated to the function `objStyle()`, so drawing the object should be like this:

```javascript
matrix.mvPushMatrix()
objStyle(**);
drawElement (**);
this.matrix.mvPopMatrix();
```

### Text rendering

Text has a very important role in rendering, text rendering in WebGL is different from OpenGL. In WebGL, the basic idea for rendering text is to build a 2D canvas, and also to create a texture by `gl.createTexture()` function. All of the text content can be created with the two dimensional canvas, like the text content, font, color, and position. Then bind the canvas image to the texture with `gl.bindTexture(gl.TEXTURE_2D,texture)` function. Then use the `gl.texParameteri(**)` function to filter the canvas image. Then bind the image to the texture. It is only needed to map the texture to the object, and then the text can be shown on the surface of the object. Here is the simple example code to describe it.

```javascript
texture = nako.gl.createTexture();
crateImage = document.getElementById('text');
image = crateImage.getContext('2d');
image.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
image.fillStyle = 'white';
image.font = "bold 36px Verdana";
text ="text content**";
text.fillText(text);
```

From the code, it first builds the texture and also gets a two dimensional canvas with `gl.createTexture()`. Then it gets the two dimensional context of the canvas, after that by using some functions or attributes which are inside the canvas. For example, the `fillStyle` attribute is the color of the text, `fillText(**)` function is to fill the text into the canvas. The next step is binding the image which is filled with text to the texture:

```javascript
nako.gl.enable(nako.gl.TEXTURE_2D);
nako.gl.bindTexture(nako.gl.TEXTURE_2D, texture);
nako.gl.texImage2D(nako.gl.TEXTURE_2D, 0, image);
nako.gl.texParameteri(nako.gl.TEXTURE_2D,nako.gl.TEXTURE_MAG_FILTER, nako.gl.LINEAR);
nako.gl.generateMipmap(nako.gl.TEXTURE_2D);
nako.gl.bindTexture(nako.gl.TEXTURE_2D, null);
```

In the code, to enable the 2D texture with `gl.enable(gl.TEXTURE_2D)`, and bind the image to the texture with the `gl.bindTexture(*)`. To filter it with `gl.texParameteri(**)` function. Now the texture with text has been built. We map the 2D texture to the object or the scene. Use the `gl.uniform1i(**)` function to activate the uniform texture variable. Set the texture uniform in the same way as we talked in the texture rendering. The simple code of binding the text texture to the object is as follows:

```javascript
nako.gl.activeTexture(nako.gl.TEXTURE0);
```
In our project, we can track the text variables, like font, position, color. To set the different text textures, use an array to get the data from the text xml. For tracking the texture uniforms, only needs to iterate the array by the name of texture uniforms.

3.4.5.5 Light rendering

Lights are created in the shader in WebGL, the attributes of the light are set with uniform variables. For example, to define a direct light, it has the location and color attributes. So we can set the two uniform variables position and color in the shader code, as discussed before. It needs to use the illumination formula to calculate the value of color, and then need to use the function gl.uniform to bind the uniform. The example code is as follows:

```javascript
namo.gl.uniform3f(uniform["color"], xml color data);
namo.gl.uniform3f(uniform["position"], xml position data);
```

The “xml color data” and “xml position data” can be got from the Document class in the namo namespace. For the single light, it is easy to build. But in the project, it is not enough to build a single light, it required to build more lights by the developers, which can be traced automatically.

To realize the trace automatically, as mentioned above, is made in the same way, to iterate the uniform array in the bind uniform function. It will be like this:

```javascript
for (x in uniform)
    namo.gl.uniform3f(uniform[x], xml data);
```

For “xml data”, we also set the same uniform array with the uniform name to store the relative data in the array. When the for loop is finished, all the light uniforms has been used.
Light rendering is not so difficult, so it is described in a simple way.

3.4.5.6 Scene visual style rendering

Scene visual style rendering is similar to object visual style rendering, but it has not shaders, it only has the element of translation, rotation, scale, and color. The render processing is similar as object visual style.
3.5 Integrate the project into OSIA project

In the background, we describe what is the OSIA project and OSIA platform. This project is not integrated to OSIA project at present, we will continue on this project in the near future.
4 Discussion

There are three phrases in this section. First we will compare the prototype with O3D [12]. Second, we evaluate the prototype in different browsers. Finally, we introduce the future work.

4.1 Comparison with O3D

This project is research and evaluate solutions 3D environment with WebGL. To evaluated the prototype, we compare it with O3D:

1. The prototype has a flexible framework. The framework is based on xml files, so it masks the nature of WebGL for web developers. For example, if we build a cube, it is very easy to create by using this prototype. The web developers only define the shape.xml and site.xml, put the vertices coordinates in the shape.xml file, define the id of the cube is “#cube”, and then imports the shape.xml into the site.xml, and the <object> includes the id of the cube. Then the cube will show on the 3D web page. The site.xml is very simple. We show the code as follow:

```xml
<?xml version="1.0" ?>
<namo>
   <import url="shape.xml" />
   <scene id="#scene1" >
      <object id="#object1" shape="#cube" />
   </scene>
</namo>
```

But in O3D, it is a little bit more complex for web developers. The web developers should first learn the API of O3D, and then use the right function to build 3D elements. How to build the cube can be learned from the O3D documentation [13].

2. To build 3D web page, our prototype is based on WebGL. It doesn’t need any plug in. But O3D need download the O3D plug-in in Google code O3D webpage [14].

3. The framework is created by XML files, the portals can be built with XML files in OSIA platform, so this prototype can be easy to integrate to OSIA project. It satisfies the requirement of the Interactive TV Arena Company. But O3D is not base the XML file framework.

4. The functions are not as rich as O3D. This project only builds a simple prototype of 3D web. The render elements are quilting few, only shape, visual style, wobble animation, camera, text and material. But O3D is powerful, it contains more render elements, like bone animation, handling events, skin and so on.

4.2 Evaluation in different browsers

The project is tested in the Chrome and Minefield browser. It works well in both browsers. It is easy to build the 3D graphics for web developers in these two browsers. The test platform is in windows 7.

The figures below show the scene rendering:
For JSON model rendering, the prototype sometimes has crashed in Minefield (it is not the final version of Firefox). It is still a problem which needs to be solved. But there is good news that Firefox 4.0 version will be published in June of 2010. The Firefox 4.0 will totally support WebGL. We hope this crash will be fixed in Firefox 4.0. The figure of the crash is shown below:
This project only realized the basic render elements, such as object, text, material, light, shading and animation rendering. It provides an interface for the developer. The developers can easily build the 3D elements, such as object, material, text, and animation with xml files. To develop more complex 3D WebPages, it is possible to combine all of these elements flexibly.

4.3 Future work

We have built a prototype 3D web browser environment for modern browsers. There are many topics that should be expanded in the future:

- To integrate the project in the OSIA project.
- To add shading that realizes bump mapping, environment mapping.
- To add bone animation
- To add video on the texture.
- To add depth shadows.

5 Conclusion

WebGL is a new technology. We have achieved most of the objectives with expected results and time constraints. First, we have designed the structure with xml files that define the model, view and control layers. The designed xml files satisfy the requirement of Interactive TV Arena. Secondly, we have built the JavaScript class to parser the structure. Thirdly, we have implemented rendering of the structure in real time with WebGL, and realized the basic elements such as shape, text, visual style, animation, material, light, shading and camera. With the XML framework, it is easy to build the 3D web page, the prototype masks the involved nature of WebGL from the web developer. We have done hard work on this project and effectively achieved most of the tasks. We suppose that with the great efforts on this project, it will be possible
to integrate with the OSIA project, and hope the OSIA platform project will be release soon successfully.

6 Acknowledgements

I wish to express sincere appreciation to Anders Willmes, the boss of interactive TV arena, for his guidance for thesis topic and his encouragement throughout the project. I would like to thank Stefan Seipel for being my supervisor, He always is positive and helpful for me, he helped me a lot with generating new ideas. He gives always positive feedback to me when it was required. And I also want to thank Fredrik Bokman, always give nice suggestions and help in this project. In particular, I want to thank Dan Kopparhed, Joakim Lofgren who was my supervisor in Interactive TV Arena Company, they always helped me a lot and give me proper time and provide their guidance. I would like to thank all teachers who gave me advice, supported throughout the study of the program of computer visualization.
Abbreviations

3D Three Dimensional
OSIA Open Server Interface Architecture
API Application programming interface
XML Extensible Markup Language
DVBT Digital Video Broadcasting Terrestrial
WebGL Web Graphics Library
ITV Independent Television
GPU Graphic Processing Unit
MVC Model–View–Controller
OpenGL Open Graphics Library
HTML5 Hyper Text Markup Language 5
GLUT the OpenGL Utility Toolkits
GLSL OpenGL Shading Language
JSON JavaScript Object Notation
Reference

Appendix A: site.xml

```
<?xml version="1.0" ?>
<namo>
  <import url="shape.xml" />
  <import url="material.xml" />
  <import url="text.xml" />
  <import url="animation.xml" />
  <import url="visual_style.xml" />
  <import url="camera.xml" />
  <import url="light.xml" />
  <import url="scene.xml" />
  <shape id="#triangle" type="TRANGLES">
    <vertices>1,1,1;0,0,0;1,0,0</vertices>
    <normals>1,0,0;1,0,0;1,0,0</normals>
    <texCoords>1,0;1,1;1,0</texCoords>
    <indices>0,1,2</indices>
  </shape>
  <material id="#material" specular="0.5" color="FFFFFF" transparence="0.5">
    <texture id="wallcolour" images/wall.jpg</texture>
  </material>
  <light id="#mainlight">
    <position>1.0,1.0,1.0</position>
    <color>#FFFFFF</color>
    <direction>1.0,1.0,1.0</direction>
  </light>
  <text id="#text1" font="font type" size="text size">
    <position>1.0,1.0,1.0</position>
    <color>#FFFFFF</color>
    <rotation>1.0,1.0,1.0</rotation>
    <textArea>input the text content</textArea>
  </text>
  <visual_style id="#visual">
    <color>#FFFFFF</color>
    <rotation angle="90">0.0,1.0,0.0</rotation>
    <scale>1.0,1.0,1.0</scale>
    <translation>1.0,1.0,1.0</translation>
    <glsl type="vertex">vert.vertex</glsl>
    <glsl type="fragment">frag.fragment</glsl>
  </visual_style>
  <camera id="#camera">
    <position>1.0,1.0,1.0</position>
    <rotation angle="90">0.0,1.0,0.0</rotation>
  </camera>
  <animation id="#animation" time="50" size="0.5">
    <direction>1.0,1.0,1.0</direction>
  </animation>
  <scene id="#scene1" camera="#camera" background color="FFFFFF">
    <object id="#object1" shape="#triangle" animate="#animation1" material="#wallmaterial" visual style="#visual0"/>
  </scene>
</namo>
```
Appendix B: document.js

document.js

Document = function() {
    this.imports = null;
    this.shape = [];
    this.material = [];
    this.camera = [];
    this.scene = [];
    this.text = [];
    this.visual_style = [];
    this.light = [];
    this.animation = [];
    this.tempObj = null;
    this.addArray = function(array, object) {
        array[object.id] = object;
    };
    this.loadXML = function(url) {
        var xhr = new XMLHttpRequest();
        xhr.open("GET", url, false);
        xhr.send(""');
        var xmlDoc = xhr.responseXML;
        var childs = xmlDoc.documentElement.childNodes;
        var len = childs.length;
        var i;
        var node;
        for (i = 0; i < len; i++) {
            node = childs[i];
            if (node.nodeType == 1) {
                switch (node.tagName) {
                    case 'shape':
                        tempObj = new shape();
                        tempObj.parse(node);
                        this.addArray(this.shape, tempObj);
                        break;
                    case 'material':
                        tempObj = new material();
                        tempObj.parse(node);
                        this.addArray(this.material, tempObj);
                        break;
                    case 'light':
                        tempObj = new light();
                        tempObj.parse(node);
                        this.addArray(this.light, tempObj);
                        break;
                    case 'text':
                        tempObj = new text();
                        tempObj.parse(node);
                        this.addArray(this.text, tempObj);
                        break;
                    case 'visual_style':
                        tempObj = new visual_style();
                        tempObj.parse(node);
                        this.addArray(this.visual_style, tempObj);
                        break;
                    case 'camera':
                        tempObj = new camera();
                        tempObj.parse(node);
                        this.addArray(this.camera, tempObj);
                        break;
                    case 'animation':
                        tempObj = new animation();
                        tempObj.parse(node);
                        this.addArray(this.animation, tempObj);
                        break;
                    case 'scene':
                        tempObj = new scene();
                        tempObj.parse(node);
                        this.addArray(this.scene, tempObj);
                        break;
                    default:
                        break;
                }
            }
        }
    }
};
default:
    alert("dont have this node");
    break;
}
}

this.load = function(url) {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, false);
    xhr.send('');
    var xmlDoc = xhr.responseXML;
    var childs = xmlDoc.documentElement.childNodes;
    var len = childs.length;
    var i;
    var node;
    for (i = 0; i < len; i++) {
        node = childs[i];
        if (node.nodeType == 1) {
            switch (node.tagName) {
            case 'import':
                this.imports = node.getAttribute('url');
                this.loadXML(this.imports);
                break;
            case 'shape':
                tempObj = new shape();
                tempObj.parse(node);
                this.addArray(this.shape, tempObj);
                break;
            case 'material':
                tempObj = new material();
                tempObj.parse(node);
                this.addArray(this.material, tempObj);
                break;
            case 'light':
                tempObj = new light();
                tempObj.parse(node);
                this.addArray(this.light, tempObj);
                break;
            case 'text':
                tempObj = new text();
                tempObj.parse(node);
                this.addArray(this.text, tempObj);
                break;
            case 'visual_style':
                tempObj = new visual_style();
                tempObj.parse(node);
                this.addArray(this.visual_style, tempObj);
                break;
            case 'camera':
                tempObj = new camera();
                tempObj.parse(node);
                this.addArray(this.camera, tempObj);
                break;
            case 'animation':
                tempObj = new animation();
                tempObj.parse(node);
                this.addArray(this.animation, tempObj);
                break;
            case 'scene':
                tempObj = new scene();
                tempObj.parse(node);
                this.addArray(this.scene, tempObj);
                break;
            default:
                break;
            }
        }
    }
};
Appendix C: process of parser xml files

Figure 13. Process of parser light

Figure 14. Process of parser animation
Figure 15. Process of parser text.xml

Figure 16. Process of parser visual_style.xml
Figure 17. Process of parser viaual_style.xml