Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 16
Tampere University of Technology. Department of Software Systems. Report 16

Matti Anttonen & Arto Salminen

Building 3D WebGL Applications

TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 16
Tampere University of Technology. Department of Software Systems. Report 16

Matti Anttonen & Arto Salminen

Building 3D WebGL Applications

Tampere University of Technology. Department of Software Systems
Tampere 2011

Matti Anttonen
matti.anttonen@tut.fi

Arto Salminen
arto.salminen@tut.fi

Department of Software Systems
Tampere University of Technology
P.O. Box 553
FIN-33101 Tampere, Finland

ISBN 978-952-15-2622-0 (printed)
ISBN 978-952-15-2623-7 (PDF)
ISSN 1797-836X

TABLE OF CONTENTS

1. INTRODUCTION

This document is a technical report of the first experimentation and evaluation phase of Lively Goes 3D project. The report is built on our experiments on WebGL and HTML5 technologies during years 2010 and 2011. The aim of this report is to give basic knowledge on how to build native 3D WebGL applications that run in web browsers. The report includes basic information about the required technologies, detailed WebGL examples that will suffice developers to get started with the new technologies and a comprehensive set for further reading and things to know about.

In Chapters 2 the main technologies used, which are HTML5 and WebGL, are introduced. Chapter 3 instructs how to build web applications with WebGL by various case examples. Further reading and other insights on the topic can be found in Chapter 4 and finally conclusions of the technical report can be found in Chapter 5.

2. BACKGROUND

Displaying 3D graphics content on the web has been possible even in the past with APIs such as Flash, O3D, VRML and X3D, but only with certain browsers or if the necessary browser plug-in components has been installed explicitly. However, with WebGL the 3D capabilities are integrated directly in the web browser, meaning that 3D content can run smoothly in any standard-compliant browser without application installation or additional components. WebGL graphics are drawn on canvas-element specified in the HTML5 specification. This Chapter gives an overview on the HTML5 and WebGL specifications that are needed for building native 3D applications for web browsers.

2.1 HTML5

HTML5 specification [1] defines the fifth version of the HTML, the central building block of web content. It replaces the current HTML4, XHTML1 and DOM Level 2 HTML (web browser JavaScript API for HTML and XHTML) standards. The specification aims to defining a single language for both XML and HTML documents, improving compatibility between web browsers, improving the web document markup and defining both markup and number of interfaces for web applications. These goals are pursued by extending the DOM interface and introducing novel APIs in addition to a number of changes to HTML elements and attributes.

New features of the HTML5 gear the web browser into a more advanced environment for web applications. Video and audio elements and APIs enable multimedia content natively, which previously has been enabled only with plug-ins. Table 1 lists the most significant features of HTML5 for the application developer.

There are also several related web technologies that have been developed together with HTML5 specification. The most important ones are listed in Table 2. These technologies are not part of either the W3C HTML5 or the WHATWG HTML specification, but W3C publishes specifications for these technologies separately.

Table 1: HTML5 features

Feature	Explanation
Canvas	Canvas element enables dynamic, scriptable rendering of two-dimensional shapes and bitmaps with low-level procedural approach.
Video and audio	Timed media playback for video and audio.
Offline	Offline storage database is meant for web applications that are able to run also when the user is offline.
ContentEditable	ContentEditable attribute makes it possible to create editable documents.
Drag-and-drop	Possibility to make draggable HTML elements.

Cross-document messaging	The new cross-document messaging system allows documents to communicate with each other in a way that does not enable cross-site scripting attacks.
History	Browser history management.
MIME type	MIME type and protocol handler registration.
Microdata	Provides a straightforward way to embed semantic information into HTML documents.

Table 2: Other forthcoming web specifications.

Feature	Explanation
Web Workers	API that allows Web application authors to spawn background workers running scripts in parallel to their main page i.e. threads.
WebGL	Enables OpenGL ES 2.0 graphics to be drawn on HTML5's canvas element.
File	API for representing and accessing file objects in web applications. Also a File Writer API is available.
WebSocket	An API that enables Web pages to use the WebSocket protocol for two-way communication with a remote host.
Indexed Database API	Indexed hierarchical key-value store (formerly WebSimpleDB).
Geolocation	Geolocation API defines a set of operations and data elements for accessing geographical location information.

2.2 WebGL

WebGL[1] is a cross-platform web standard for hardware accelerated 3D graphics API developed by WebGL Working Group, which includes many industry leaders such as AMD, Apple, Ericsson, Google, Mozilla, NVIDIA and Opera. WebGL brings the ability to display 3D graphics natively in the web browser without any plug-in components.

Moreover, even though WebGL is designed for 3D graphics in mind, it can be used for 2D graphics as well. WebGL is based on OpenGL ES 2.0[2], and it uses the OpenGL shading language GLSL [3]. WebGL runs in the HTML5 canvas element, and WebGL

[1] http://www.khronos.org/webgl/
[2] http://www.khronos.org/opengles

objects are generally accessible through the web browser's Document Object Model (DOM) interfaces. A comprehensive JavaScript API is provided to open up OpenGL programming capabilities to web application developers.

Displaying 3D graphics content on the web has been possible in the past with APIs such as Flash, O3D, VRML and X3D, but only with certain browsers or if the necessary browser plug-in components has been installed explicitly. However, with WebGL the 3D capabilities are integrated directly in the web browser, meaning that 3D content can run smoothly in any standard-compliant browser without application installation or additional components.

The final 1.0 version of the specification was released on 3rd March [2]. WebGL support has already been implemented in the latest versions of Google Chrome (Version 9) and it is to be included in the forthcoming new versions of Apple Safari, Mozilla Firefox and Opera.

Five years back the JavaScript performance would have not be enough for running more complicated WebGL examples but fortunately the situation have got much better as seen in Figure 1 taken from the Google's Chromium blog[3].

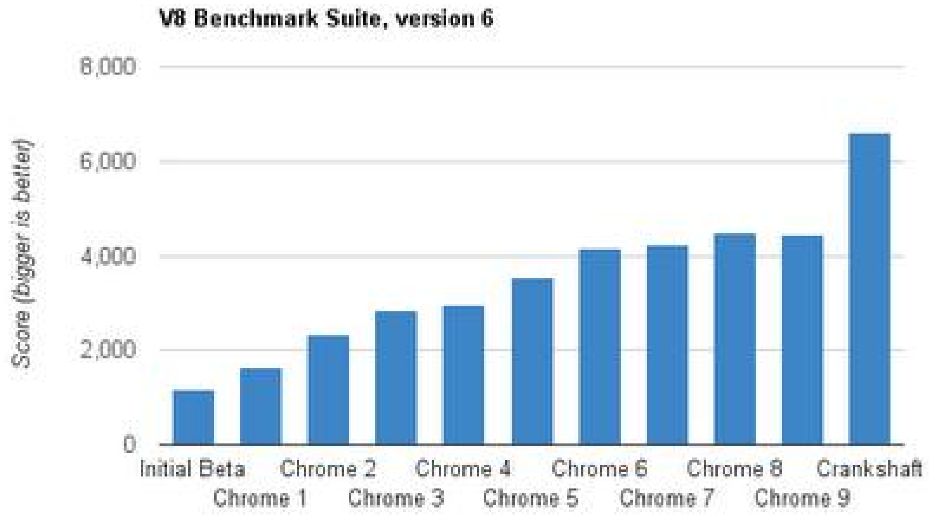

Figure 1: Improvement in JavaScript performance.

The WebGL API is implemented at a lower level compared to the equivalent OpenGL APIs. This increases the software developers' burden as they have to implement some commonly used OpenGL functionality themselves. To make it easier and faster to use WebGL, several independent middle layer WebGL frameworks have been developed.

All the frameworks above have their own JavaScript API through which the actual WebGL API is used. In general, the goal of these libraries is to hide the majority of technical details and thus make it simpler to write applications using the framework

[3] http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html

APIs. Furthermore, WebGL can be used in combination with existing JavaScript libraries and toolkits such as Dojo[4], jQuery[5] and Prototype[6].

The most useful formats for using more complex 3D models in your WebGL applications are JSON and Collada at the moment. Some 3D graphic tools support exporting to JSON format which can be easily loaded into JavaScript by using libraries such as json2.js[7]. Some WebGL frameworks also support Collada format[8] directly. Collada files can be found and shared for example in Google Warehouse[9], OurBricks[10] and Kuda[11].

2.2.1 C3DL

Developer: CATGames Research network

Version: 2.2 (released 22 March, 2011)

C3DL started as a middle layer API for Canvas 3D, the precursor of WebGL. The version 2.0, which was released on February 22nd, updated all the core functionalities of C3DL to use WebGL. Versions 2.1 and 2.2 have not introduced many new features but they have improved the existing functionality and made it more stable.

C3DL distinguishes from the other WebGL frameworks by its very good and instructive tutorials. On top of describing how the library is used the tutorials cover the basics of 3D graphics as well.

Supported 3D models: Collada

Website: http://www.c3dl.org/

2.2.2 Copperlicht

Developer: Ambiera e.U.

Version: 1.3.7 (released 23 June, 2011)

What makes Copperlicht different from the other WebGL developer libraries is that it is developed to be used with a 3D world editor of its own, called CopperCube. The editor makes it possible to compress 3D meshes into binary files to make it faster for Copperlicht to load them. Even though Copperlicht is free to use CopperCube is under a commercial license.

A game development company called Ambiera is developing both the editor and the library actively. The latest version includes support for character animations.

From the performance point of view Copperlicht has been one of the fastest WebGL libraries available. Even bigger demos and 3D models seem to run pretty smoothly when tested with Chrome.

[4] http://www.dojotoolkit.org/
[5] http://jquery.com
[6] http://www.prototypejs.org
[7] http://www.json.org/
[8] https://collada.org
[9] http://sketchup.google.com/3dwarehouse/
[10] http://beta.ourbricks.com
[11] http://code.google.com/p/kuda/

Supported 3D models: 3ds, obj, x, lwo, b3d, csm, dae, dmf, oct, irrmesh, ms3d, my3D, mesh, lmts, bsp, md2, stl, ase, ply, dxf, cob, scn and more

Website: http://www.ambiera.com/copperlicht/

2.2.3 CubicVR.js

Developer: Charles J. Cliffe

Version: -

CubicVR.js is a JavaScript port of C++ engine called CubicVR. For documentation there are only a few demos available, some of them demonstrates a nice feature called BeatDetector, which is a music visualizing application using HTML5 audio.

Supported 3D models: -

Website: https://github.com/cjcliffe/CubicVR.js

2.2.4 EnergizeGL

Developer: Denny Koch

Version: -

One of the newcomers in the WebGL developer libraries is EnergizeGL, which was published in March 2010. The current version is still at the very beginning. It only supports drawing basic 2D and 3D shapes on the canvas. EnergizeGL is also using an old matrix calculations library called sylvester.js so there is still much performance optimization to be done on the framework.

Supported 3D models: none

Website: http://energize.cc/

2.2.5 GammaJS

Developer: Stephen Moore, Royce Townsend and James Campbell

Version: 0.1

Gamma is designed for creating 2.5D platform games for a web browser using HTML, JavaScript, CSS and WebGL. It offers basic features for building platform games and most of the features are demonstrated by examples.

Supported 3D models:

Website: http://gammajs.org/

2.2.6 GLGE

Developer: Paul Brunt

Version: 0.8

Perhaps the most actively developed WebGL developer library so far has been GLGE. New features have been added to it steadily after the initial release and at the moment the library supports even some more advanced features like fog and depth shadows. The availability of different features has given GLGE an edge compared to most other WebGL libraries.

Downside of GLGE is that there are no tutorials available for the library as only simple demos of all the main features are shown at the webpage.

Supported 3D models: Collada

Website: http://www.glge.org/

2.2.7 J3D

Developer: Bartek Drozdz

Version:

J3D is a JavaScript/WebGL engine created by the author for his own purposes. The source code is available on github and examples can be found from the developer's home page. This library should not be mixed with a Java 3D engine called J3D.

Supported 3D models: -

Website: https://github.com/drojdjou/J3D

2.2.8 Jax

Developer: Colin MacKenzie IV

Version: 1.1.0

Jax is a WebGL framework that is intended to be used with Ruby. On top of basic features Jax also has a plugin system, which makes it easier to develop reusable moduls that are easy the share as well. The whole framework and documentation for it are under heavy but active development.

Supported 3D models: -

Website: http://blog.jaxgl.com/

2.2.9 O3D

Developer: Google

Version: 20100829

O3D was originally a plugin for Google Chrome that made it possible to render 3D graphics in the browser. When WebGL was published a WebGL implementation of O3D was published as well. The support for this framework has stopped and the downloadable zip-file includes all the Google's demos for the framework. Google did a pretty good job on optimizing many things in O3D and some parts of the project have been ripped out and "released" separately.

Project website also offers source for a Collada-to-JSON converter to make it possible to import own meshes.

Supported 3D models: Collada

Website: http://code.google.com/p/o3d/

2.2.10 osgjs

Developer: Cedric Pinson

Version: -

osgjs is an implementation of C++ OpenSceneGraph, which is an open source high performance 3D graphics toolkit. It offers “OpenSceneGraph-like” toolbox to interact with OpenGL via JavaScript, and provides facilities for exporting various assets to the osgjs format.

Supported 3D models: osgjs, the project homepage offers a link in which you can transform your own 3D models into osgjs.

Website: http://osgjs.org/

2.2.11 PhiloGL

Developer: Nicolas Garcia Belmonte

Version: 1.3.0

PhiloGL is specialized in data visualization in 3D, but it can also be used for anything else as well. The framework’s tutorials complete the same examples as in learning-webgl.com, which makes it easy to compare the framework to pure WebGL.

Supported 3D models: JSON

Website: http://senchalabs.github.com/philogl/

2.2.12 SceneJS

Developer: Lindsay Kay, Xeolabs

Version: 0.8.0 Beta

SceneJS seems to have started as a research project that has later on turned into a more product-like library. On top of the basic 3D functionalities SceneJS supports LOD (level of detail). The project website got a well explained playroom-section, where one can try and modify SceneJS examples online.

The development of the library has been steady and the core of the framework is well optimized, which makes the whole library very promising.

Supported 3D models: Collada

Website: http://scenejs.org/

2.2.13 SpiderGL

Developer: Marco Di Benedetto, A Visual Computing Laboratory - ISTI

Version: 0.1.1.20100129

After the release of version 0.1.1 (January 29th) there has been no real updates available for the SpiderGL library itself. Later on the development of the library has stopped at the current version, released 13th December 2010. SpiderGL have got some nice features like LOD, basic reflections, particles and shadows working. The downside seems to be that loading big Collada files to the canvas take some time.

Supported 3D models: Collada

Website: http://spidergl.org/

2.2.14 TDL

Developer: Gregg Tavares

Version: -

TDL is a low-level library for WebGL apps. It currently focuses on speed of rendering rather than ease of use.

Supported 3D models: -

Website: http://code.google.com/p/threedlibrary/

2.2.15 three.js

Developer: Mr.doob

Version: r44

three.js is one of the most promising WebGL frameworks available. The framework itself claims to be designed for dummies but the lack of documentation makes it hard to get started with. The smooth and visually beautiful demos look very promising and the number of other developers using the library proves that the library have already found its target audience. Unlike most of these frameworks, three.js can render on either <canvas>, <svg> or WebGL.

Supported 3D models: Collada

Website: https://github.com/mrdoob/three.js

2.2.16 WebGLU

Developer: Benjamin P. DeLillo

Version: -

WebGLU was the first WebGL developer library to be publicly available. It was published in the last quarter of 2009 with a few features and has been slowly developed further.

On one hand WebGLU benefits from the status of been the first developer library to be published but on the other hand the pace of the development has been so slow that it currently only support the main animation functionalities and shader import features.

The only official information available about WebGLU is found at developer’s blog. Even the API documentation is only available inside the downloadable archive.

Supported 3D models: .frag, .vert, .vp, .fp, .obj (partially)

Website: http://github.com/OneGeek/WebGLU

2.2.17 x3dom

Developer: Fraunhofer IGD

Version: 1.2

x3dom has been developed to fulfill the HTML5 specification for declarative 3D content as it allows including X3D elements as part of any HTML5 DOM tree. The coding is much more done in HTML and CSS way than in JavaScript, which might be easier approach for developers not familiar with JavaScript.

Supported 3D models: X3D. Exporters available for Blender, Maya, 3ds Max and WOWModelViewer

Website: http://www.x3dom.org/

3. BUILDING WEBGL APPLICATIONS

This chapter provides detailed examples on how to build native 3D WebGL web applications in either pure WebGL or with the help of GLGE framework[12]. GLGE framework is selected from the many WebGL utility frameworks because it has been developed steadily, it has implemented all the most needed features and it is sufficiently documented online[13]. GLGE also proved out to be the most promising free WebGL framework in our earlier project report [4].

To follow the examples basic knowledge of JavaScript are needed. To build 3D web applications the basics of drawing 3D graphics are also required. Any experiences with OpenGL or GLSL shading language are also useful as WebGL bases on OpenGL ES 2.0, but not obligatory. For further reading on JavaScript Douglas Crockford's book JavaScript: The Good Parts [5] is an excellent resource. More information about 3D graphics can be found for example in Donald Hearn's and Pauline Baker's book Computer Graphics with OpenGL [6].

No other installation or development environment is needed for testing these examples than a WebGL capable web browser and a web server to deploy the web application into. An up-to-date list of WebGL capable web browsers can be found online: http://www.khronos.org/webgl/wiki/Getting_a_WebGL_Implementation

3.1 WebGL Examples

Five WebGL examples are described in detail in this subchapter. The first example goes through the needed core WebGL code for displaying simple 2D objects on the screen. The second example expands the first by adding color to the objects. The third example shows how objects can be rotated and the fourth example transforms the objects into real 3D objects. The fifth example adds support for textures.

Examples are based on Learning WebGL blog's lessons[14] that are furthermore based on NeHe's OpenGL tutorials[15].

All the function calls in the code that starts with prefix gl. refers to calling WebGL API's functions. Full API documentation with descriptions for these functions can be found in the WebGL Technical Specification [2]. All the other functions used are either implemented in the example or imported from the used JavaScript math and matrix libraries called Sylvester.js[16] and glUtils.js[17] which augments the Sylvester.js library with a few utility functions.

[12] http://www.glge.org
[13] http://www.glge.org/api-docs
[14] http://learningwebgl.com
[15] http://nehe.gamedev.net/
[16] http://sylvester.jcoglan.com
[17] http://code.google.com/p/gew/source/browse/trunk/Sample/Utils/glUtils.js?r=13

3.1.1 Displaying basic 2D shapes

In the first example two basic 2D shapes, a triangle and a square, are drawn on the canvas-element. This example demonstrates all the obligatory parts of code needed for setting up a simple WebGL application.

Listing 1 shows the HTML code containing the HTML5 canvas element.

```
<!DOCTYPE html>
<html>
 <head>
     <title>WebGL example 1</title>
     <meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
     <script type="text/javascript" src="sylvester.js"></script>
     <script type="text/javascript" src="glUtils.js"></script>
 </head>
 <body onload="webGLStart();">
     <canvas id="example-canvas" style="border: none;"
     width="500" height="500"></canvas>
 </body>
</html>
```

Listing 1. HTML code for the first WebGL example.

After the HTML5's <!DOCTYPE html> definition two utility libraries are loaded. Sylvester.js is a JavaScript math library for handling vectors and matrices. glUtils.js augments Sylvester.js with functions needed for perspective 3D drawing. Both libraries are available online for free and are widely used with WebGL examples.

The <body> section includes <canvas> element in which the WebGL content is rendered. Canvas element is labeled to make it easier to access it via JavaScript's DOM functions and it is initialized to size 500x500 pixels. When the <body> section of the web page is loaded a JavaScript function called webGLStart() is called.

To draw something in the canvas three more JavaScript scripts have to be added to the <head> section of the HTML code. Firstly two shaders are needed for drawing the output. In short explanation a Vertex shader is called for every vertex the application is drawing and it defines where the vertex is drawn and a Fragment shader defines the color for every pixel to be drawn. The shader codes are actually run on the graphic card, which significantly helps JavaScript performance. Shaders used in the first example can be seen in Listing 2 and although they syntactically look like JavaScript they are actually written in GLSL shading language [6].

```
<script id="shader-fs" type="x-shader/x-fragment">
    #ifdef GL_ES
    precision highp float;
    #endif
    void main(void) {
        gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
</script>
```

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
  }
</script>
```

Listing 2. Fragment and vertex shaders used in the first WebGL example.

The #ifdef part of the fragment shader tells the graphic card what float precision to use. In this first example everything is drawn in white, so the Fragment shader simply defines the drawing color for every pixel to be white by giving it RGB values and an alpha parameter for defining transparency.

The vertex shader defines one attribute (attributes are the input of the shader), aVertex-Position, which is a three dimension vector for defining the coordinates in which the current vertex is located in a 3D space. Also two uniforms are defined: a model-view matrix and a projection matrix, which will be used for calculating 3D projections for the current vertex. The Vertex shader is called for every vertex in the scene and the main-function calculates where the given vertex should be drawn in the canvas.

After we have defined the shaders we are ready to write the actual application, which is done by adding the third JavaScript script into the <head> section. In this script we need to define the webGLStart() function (seen in Listing 3) that is called when the webpage is loaded.

The webGLStart() function simply initializes everything and draws the application into screen. First the <canvas> element is initialized for handling WebGL context (initGL), the shaders we just created are initialized (initShaders) and the triangle and the square we have decided to draw are created (initBuffers). After that a clearing color for the canvas is defined to be black (gl.clearColor) and something called a depth test is enabled. Enabling the depth test hides objects drawn behind objects in front of them. Finally a function that draws these shapes into the <canvas> element is called (drawS-cene()).

The webGLStart() function is seen in Listing 3.

```
function webGLStart() {
    var canvas = document.getElementById("example-canvas");
    initGL(canvas);
    initShaders();
    initBuffers();

    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.enable(gl.DEPTH_TEST);

    drawScene();
}
```

Listing 3. webGLStart() function.

Next the initialization functions initGL(), initShaders() and initBuffers() have to be created. The initGL function that initializes WebGL context can be seen in Listing 4.

```
var gl;
function initGL(canvas) {
    try {
      gl = canvas.getContext("experimental-webgl");
      gl.viewport(0, 0, canvas.width, canvas.height);
    } catch(e) {
    }
    if (!gl) {
      alert("Could not initialise WebGL, sorry :-(");
    }
}
```

Listing 4. webGLStart() function.

Before the initGL function a global variable called gl is created. The WebGL context is saved to this variable by calling <canvas> element’s getContext function with a parameter “experimental-webgl”. When the getContext function is called with this parameter a WebGLRenderingContext object is created and returned to gl-variable and functions defined in the WebGL specification can be called by using the syntax gl.functionName(parameters). The “experimental-webgl” parameter has been used with the current web browsers but it should be changing into “webgl” in the future browser versions as the WebGL specification has reached version 1.0. After that we tell the WebGL context the size of our canvas (gl.viewport) and check if WebGL context got initialized correctly or not.

For initializing the shaders we need to make two functions: initShaders() seen in Listing 5 and getShader() seeing in Listing 6.

```
var shaderProgram;
function initShaders() {
  var fragmentShader = getShader(gl, "shader-fs");
  var vertexShader = getShader(gl, "shader-vs");

  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)){
    alert("Could not initialise shaders");
  }
  gl.useProgram(shaderProgram);
  shaderProgram.vertexPositionAttribute=gl.getAttribLocation(shaderProgram,
                                                              "aVertexPosition");
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
  shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram,"uPMatrix");
  shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram,"uMVMatrix");
}
```

Listing 5. Code for initShaders() function.

The initShaders() function finds the fragment and vertex shaders we created by calling getShader() function and creates a global variable called shaderProgram, which is initialized as a WebGLProgram object (gl.createProgram). Each WebGLProgram object can hold two shaders, a fragment shader and a vertex shader, which it will run on the graphics card. After the shaderProgram variable is initialized, the shaders are attached to it (gl.attachShader), the object is linked (gl.linkProgram) and finally it is set to be used (gl.useProgram). As we created an aVertexPosition attribute into the vertex shader (seen in Listing 2) we can now save it to the shaderProgram as well (gl.getAttribLocation) as we need it for passing the triangle's and square's vertices to the shader. We also need to tell WebGL that we want to provide values for the attribute by using this array (gl.enableVertexAttribArray). The last two lines store the model-view and the projection matrices created in the vertex shader (seen in Listing 2) to the shaderProgram as well (gl.getUniformLocation). Now the shaders are initialized to the graphics card and are ready to be used.

```
function getShader(gl, id) {
    var shaderScript = document.getElementById(id);
    if (!shaderScript) {
      return null;
    }

    var str = "";
    var k = shaderScript.firstChild;
    while (k) {
      if (k.nodeType == 3) {
        str += k.textContent;
      }
      k = k.nextSibling;
    }

    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
      shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
      shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
      return null;
    }

    gl.shaderSource(shader, str);
    gl.compileShader(shader);
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert(gl.getShaderInfoLog(shader));
      return null;
    }

    return shader;
}
```

Listing 6. Code for getShader() function.

The getShader() function used in the initShaders() function is seen in Listing 6. The code finds the wanted element in the HTML code defined by the element's id and saves textContent from all the elements's text nodes (nodeType == 3) into a single string. It is then checked whether a fragment or a vertex shader is to be created and the created string is used to initialize the shader. After that the shader is compiled (gl.compileShader) into a form that is understood by the WebGLProgram object used in the initShaders() function. The created shader is returned as an object.

After the WebGL context and shaders are initialized and we can start drawing objects. First we have to create the triangle and the square objects we want to display and to write a function that draws them on the canvas. The objects are defined in an initBuffers() function by defining the bounds of the objects by vertices as seen in Listing 7.

```
var triangleVertexPositionBuffer;
var squareVertexPositionBuffer;
function initBuffers() {
  triangleVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
  var vertices = [
         0.0,  1.0,  0.0,
        -1.0, -1.0,  0.0,
         1.0, -1.0,  0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  triangleVertexPositionBuffer.itemSize = 3;
  triangleVertexPositionBuffer.numItems = 3;

  squareVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
  vertices = [
         1.0,  1.0,  0.0,
        -1.0,  1.0,  0.0,
         1.0, -1.0,  0.0,
        -1.0, -1.0,  0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  squareVertexPositionBuffer.itemSize = 3;
  squareVertexPositionBuffer.numItems = 4;
}
```

Listing 7. initBuffers() function.

The initBuffer() function creates two buffers (gl.createBuffer), triangleVertexPosition-Buffer for the triangle and squareVertexPositionBuffer for the square. The buffers are bound to the WebGL context (gl.bindBuffer) and needed vertices (three needed to draw a triangle and four needed to draw a square) are added to the buffer (gl.bufferData). The vertices are given as coordinates, which defines the corner points of the objects in a 3D space. Two parameters are passed to both buffers: itemSize, which indicates the length of each coordinate and numItems, which defines how many coordinates each buffer holds (three for triangle, four for square). These parameters are needed when the objects are drawn into the canvas in the drawScene() function shown in Listing 8.

```
function drawScene() {
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);


  perspective(45, 1.0, 0.1, 100.0);
  loadIdentity();
  mvTranslate([-1.5, 0.0, -7.0]);

  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
               triangleVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
  setMatrixUniforms();
  gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.numItems);

  mvTranslate([3.0, 0.0, 0.0]);
  gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                squareVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
  setMatrixUniforms();
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, squareVertexPositionBuffer.numItems);
}
```

Listing 8. drawScene() function.

The drawScene() function starts with clearing the whole canvas element with the clearing color we defined in webGLStart() function. Next we will set the perspective from which we want to view the scene. As default WebGL draws things in orthographic projection but as we want to use perspective projection so we need to set the perspective. In this scene we will set the field of view to 45°, width-to-height ratio to 1.0 (as our canvas was set to 500x500 pixels), near clipping plane to 0.1 and far clipping plane to 100.0.

Defining a position in 3D space can be done using a 4x4 matrix which will hold the coordinates of the point and a vector stating which way we are looking. After this all the needed transitions (moving the camera, moving the objects, finding a certain position etc.) can be done by using matrix calculations. Two matrices are usually used for this: a model-view matrix for placing objects in the right place in the scene and a projection matrix which holds the position of the camera.

To draw our triangle in the desired place in the scene we will first have to initialize our model-view matrix to identity matrix to find the origin. After that we will translate the coordinates 1.5 steps to the left and 7 steps into the scene (further from the point of view) with a matrix calculation (mvTranslate). As we have found the correct place we will draw the triangle by binding its buffer to the WebGL context (gl.bindBuffer), assigning its vertex buffer to the vertex shader (gl.vertexAttribPointer), taking account of our current model-view and perspective matrices (setMatrixUniforms) and calling gl.drawArrays, which will finally draw the triangle on the scene. After that we repeat the same steps for the square. As we want to draw the square next to the triangle we do not have to reset our model-view matrix to identity matrix in between. Instead we can use the current position stored in the model-view matrix and simply translate 3 steps to the right to find the correct position for the square. The same result would be gained if we first called loadIdentity to get back to origin and then translated 1.5 steps to right and seven steps into the scene.

The first parameter of the gl.drawArrays call defines the way the vertices are drawn. With the triangle we simply give gl.TRIANGLES, which means that the given vertices are drawn simply as triangles. For the cube we use a different parameter gl.TRIANGLE_STRIP, which is a strip of triangles where the first three vertices you give specify the first triangle, and then the last two of those vertices plus the next one specifies the next triangle, and so on.

To finalize our example we still need to define our model-view and projection matrices and write code for those few matrix functions used in drawScene function. The needed functions are seen in Listing 9.

```
var mvMatrix;
function loadIdentity() {
  mvMatrix = Matrix.I(4);
}

function multMatrix(m) {
  mvMatrix = mvMatrix.x(m);
}

function mvTranslate(v) {
  var m = Matrix.Translation($V([v[0], v[1], v[2]])).ensure4x4();
  multMatrix(m);
}

var pMatrix;
function perspective(fovy, aspect, znear, zfar) {
  pMatrix = makePerspective(fovy, aspect, znear, zfar);
}

function setMatrixUniforms() {
  gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false,
                      new Float32Array(pMatrix.flatten()));
  gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false,
                      new Float32Array(mvMatrix.flatten()));
}
```

Listing 9. Needed matrix functions.

These functions create the model-view (mvMatrix) and perspective (pMatrix) matrices for our application and use the Sylvester.js and glUtils.js libraries for the needed matrix calculations. The setMatrixUniforms function sends the values of our JavaScript matrices to the vertex shader's uniforms.

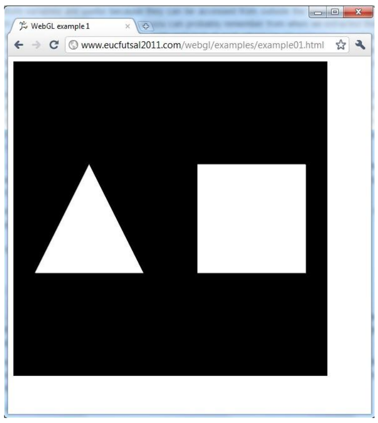

Figure 2: Example 1 seen in Chrome.

Screenshot of the resulting WebGL application can be seen in Figure 2. Full source code for this first example can be found at Attachment 1.

3.1.2 Adding colors

In the second example we will add some color to the existing shapes. Adding colors to the existing example is pretty straightforward. First we need to modify our shaders so that they can draw different colors, then we will have to add color buffers to our shapes to define in which color to draw each object and finally the drawScene function needs to be updated so that the added color buffers are used in the drawing. The modified shader codes can be seen in listing 10.

```
<script id="shader-fs" type="x-shader/x-fragment">
  #ifdef GL_ES
  precision highp float;
  #endif

  varying vec4 vColor;

  void main(void) {
    gl_FragColor = vColor;
  }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;
  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  varying vec4 vColor;

  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vColor = aVertexColor;
  }
</script>
```

Listing 10. Shader codes for the second example.

What we have done here is that we have added another attribute (attributes were the input of the shader) to the vertex shader called aVertexColor. So in addition to calculating the position of the vertex to be drawn we can also specify the color in which it should be drawn. We have also added one varying (these are the output of the shader) called vColor into which we simply assign the given input color (aVertexColor).

In the fragment shader we will simply add one varying called vColor, which will be directly used as the color of the pixel. The value of the vColor is interpolated automatically for each pixel as the fragment shader is called for every pixel.

As we added the aVertexColor attribute to the vertex shader we need to add a reference to it in the initShaders function as well. This is done by adding lines shown in Listing 11 to the initShaders function.

```
shaderProgram.vertexColorAttribute = gl.getAttribLocation(shaderProgram,
                                                          "aVertexColor");
gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
```

Listing 11. Addition to the initShaders() function.

This will add a vertexColorAttribute to the shaderProgram, map it to the aVertexColor attribute we added to the vertex shader and enable the attribute in the WebGL context. The shaders are now ready for displaying colors.

Next we need to add color buffers to the triangle and the square. This is done by adding lines shown in Listing 12 to the initBuffers.

```
triangleVertexColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
var colors = [
        1.0, 0.0, 0.0, 1.0,
        0.0, 1.0, 0.0, 1.0,
        0.0, 0.0, 1.0, 1.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
triangleVertexColorBuffer.itemSize = 4;
triangleVertexColorBuffer.numItems = 3;

squareVertexColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexColorBuffer);
colors = []
for (var i=0; i < 4; i++) {
  colors = colors.concat([0.5, 0.5, 1.0, 1.0]);
}
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
squareVertexColorBuffer.itemSize = 4;
squareVertexColorBuffer.numItems = 4;
```

Listing 12. Addition to the initBuffers() function.

This addition specifies color buffers for both objects in the same way as their vertices were specified in the previous example. The colors are given with four parameters: red, green, blue and alpha, which defines which color is used in each vertex.

Finally we have to add a few lines into the drawScene() function to get the colors into effect as seen in Listing 13.

```
gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
                      triangleVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);

gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexColorBuffer);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
                      squareVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

Listing 13. Addition to the drawScene() function.

The first gl.bindBuffer and gl.vertexAttribPointer calls are for drawing the triangle so they need to be added just before setMatrixUniforms() is called for the first time in the drawScene() function. The second gl.bindBuffer and gl.vertexAttribPointer calls are for drawing the triangle so they need to be added just before setMatrixUniforms() is called for the second time. Screenshot of the resulting application is seen in Figure 3.

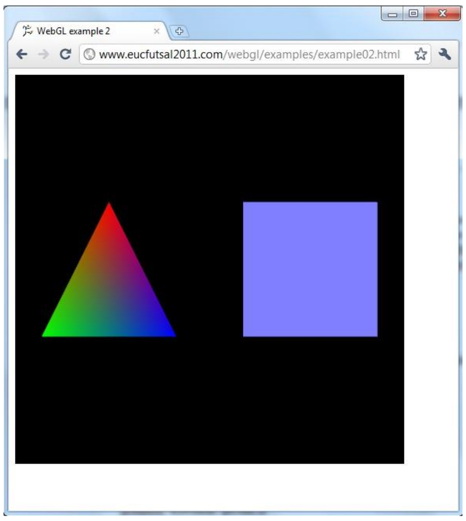

Figure 3: WebGL example with colors.

3.1.3 Adding rotation

In our third example we will make the triangle and cube rotate around different axis. In our current example the drawScene() function was simply called once in the end of the webGLStart() function. To animate rotation we will have to make a timer in our application, which will repeatedly invoke a function that will calculate the rotation for the next frame and call the drawScene function again.

To start with, the direct call of drawScene() in the end of webGLStart() have to be replaced by the first line shown in Listing 14, which will set a JavaScript timer that will call a function called tick() every 10 milliseconds. Also functions called tick() and animate() as seen in the Listing 14 have to be added into the script.

```
setInterval(tick, 10);
function tick() {
  drawScene();
  animate();
}

var lastTime = 0;
var rTri = 0;
var rSquare = 0;
function animate() {
  var timeNow = new Date().getTime();
  if (lastTime != 0) {
    var elapsed = timeNow - lastTime;
    rTri += (90 * elapsed) / 1000.0;
    rSquare += (75 * elapsed) / 1000.0;
  }
  lastTime = timeNow;
}
```

Listing 14. Functions to be added for calculating rotation.

The tick() function does two function calls. First it calls drawScene() function to draw the objects into the scene and then it will call animate() function, that will calculate the rotation of the objects for the next frame. The animate() function creates needed variables, calculates how much time have elapsed since the function was called the last time and updates the rotations (rTri and rSquare) based on the elapsed time.

In the previous examples we were simply drawing once and as the objects were stationary we came along nicely with only using one model-view matrix. As we are starting to animate multiple objects it is easier if after drawing one object at certain coordinates with certain rotation one could navigate back to some specific location in the scene and then make the transitions and rotations for the next object. This is possible by making a stack of the model-view matrices so that each transition can be reversed. To make this possible we need to add a variable called mvMatrixStack and two functions for handling the stack: mvPushMatrix() and mvPopMatrix(). Also two more matrix functions are needed for handling the matrix calculations need for rotations. The added functions are shown in Listing 15.

```
var mvMatrixStack = [];
function mvPushMatrix(m) {
  if (m) {
    mvMatrixStack.push(m.dup());
    mvMatrix = m.dup();
  } else {
    mvMatrixStack.push(mvMatrix.dup());
  }
}

function mvPopMatrix() {
  if (mvMatrixStack.length == 0) {
    throw "Invalid popMatrix!";
  }
  mvMatrix = mvMatrixStack.pop();
  return mvMatrix;
}

function createRotationMatrix(angle, v) {
  var arad = angle * Math.PI / 180.0;
  return Matrix.Rotation(arad, $V([v[0], v[1], v[2]])).ensure4x4();
}

function mvRotate(angle, v) {
  multMatrix(createRotationMatrix(angle, v));
}
```

Listing 15. Added matrix functions.

The final thing to do is to update the drawScene function to use the matrix stack and to rotate the objects. The updated drawScene() function can be seen in Listing 15.

```
function drawScene() {
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  perspective(45, 1.0, 0.1, 100.0);
  loadIdentity();

  mvPushMatrix();
  mvTranslate([-1.5, 0.0, -7.0]);
  mvRotate(rTri, [0, 1, 0]);

  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                triangleVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
                triangleVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);
  setMatrixUniorms();
  gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.numItems);

  mvPopMatrix();
  mvPushMatrix();
  mvTranslate([1.5, 0.0, -7.0]);
  mvRotate(rSquare, [1, 0, 0]);

  gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                 squareVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
  gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexColorBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
                 squareVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);
  setMatrixUniforms();
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, squareVertexPositionBuffer.numItems);

  mvPopMatrix();
}
```

Listing 16. drawScene() function with rotation.

The drawScene() code is otherwise the same as in the previous example, but we have just added the calls needed for using the stack and rotating the objects. In the previous example we first navigated to the coordinates of the triangle and then shifted three steps to right to draw the square. This time we modified the drawing code so that after drawing the triangle we return to origin (calling mvPopMatrix() and mvPushMatrix()) and then make the needed matrix translations before drawing the square. In other words, even though the square is displayed at the same coordinates than before, it is not anymore related three steps right from the triangle, instead it is located 1.5 steps right and 7 steps into the scene from the origin, which can be seen from the parameters of the mvTranslate() function call. To make the objects rotate we have called mvRotate after the mvTranslate calls. When called in this order the objects will rotate around themselves instead of rotating around the whole Y- or Z-axis as seen in Figure 4.

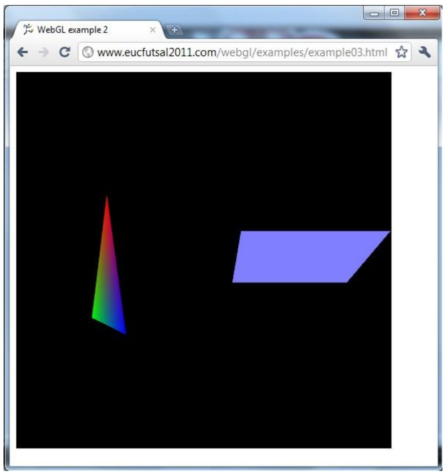

Figure 4: WebGL example with rotation.

3.1.4 Adding third dimension

Changing the objects in our example into third dimension can be done by defining a cube with six squares and a pyramid with four triangles and drawing them in appropriate positions in 3D space to create 3D objects. This way we only have to define more vertices and colors in the initBuffer function. All the updated vertices and colors for the pyramid are listed in Listing 17 and for the cube in Listing 18.

```
var vertices = [
        // Front face
         0.0,  1.0,  0.0,
        -1.0, -1.0,  1.0,
         1.0, -1.0,  1.0,
        // Right face
         0.0,  1.0,  0.0,
         1.0, -1.0,  1.0,
         1.0, -1.0, -1.0,
        // Back face
         0.0,  1.0,  0.0,
         1.0, -1.0, -1.0,
        -1.0, -1.0, -1.0,
        // Left face
         0.0,  1.0,  0.0,
        -1.0, -1.0, -1.0,
        -1.0, -1.0,  1.0
];
```

```
var colors = [
        // Front face
        1.0, 0.0, 0.0, 1.0,
        0.0, 1.0, 0.0, 1.0,
        0.0, 0.0, 1.0, 1.0,
        // Right face
        1.0, 0.0, 0.0, 1.0,
        0.0, 0.0, 1.0, 1.0,
        0.0, 1.0, 0.0, 1.0,
        // Back face
        1.0, 0.0, 0.0, 1.0,
        0.0, 1.0, 0.0, 1.0,
        0.0, 0.0, 1.0, 1.0,
        // Left face
        1.0, 0.0, 0.0, 1.0,
        0.0, 0.0, 1.0, 1.0,
        0.0, 1.0, 0.0, 1.0
];
```

Listing 17. Vertices and colors for the pyramid.

```
vertices = [
      // Front face
      -1.0, -1.0,  1.0,
       1.0, -1.0,  1.0,
       1.0,  1.0,  1.0,
      -1.0,  1.0,  1.0,

      // Back face
      -1.0, -1.0, -1.0,
      -1.0,  1.0, -1.0,
       1.0,  1.0, -1.0,
       1.0, -1.0, -1.0,

      // Top face
      -1.0,  1.0, -1.0,
      -1.0,  1.0,  1.0,
       1.0,  1.0,  1.0,
       1.0,  1.0, -1.0,

      // Bottom face
      -1.0, -1.0, -1.0,
       1.0, -1.0, -1.0,
       1.0, -1.0,  1.0,
      -1.0, -1.0,  1.0,
      // Right face
       1.0, -1.0, -1.0,
       1.0,  1.0, -1.0,
       1.0,  1.0,  1.0,
       1.0, -1.0,  1.0,

      // Left face
      -1.0, -1.0, -1.0,
      -1.0, -1.0,  1.0,
      -1.0,  1.0,  1.0,
      -1.0,  1.0, -1.0,
];

colors = [
      [1.0, 0.0, 0.0, 1.0],     // Front face
      [1.0, 1.0, 0.0, 1.0],     // Back face
      [0.0, 1.0, 0.0, 1.0],     // Top face
      [1.0, 0.5, 0.5, 1.0],     // Bottom face
      [1.0, 0.0, 1.0, 1.0],     // Right face
      [0.0, 0.0, 1.0, 1.0],     // Left face
];
var unpackedColors = [];
for (var i in colors) {
  var color = colors[i];
  for (var j=0; j < 4; j++) {
    unpackedColors = unpackedColors.concat(color);
  }
}
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(unpackedColors), gl.STATIC_DRAW);
squareVertexColorBuffer.itemSize = 4;
squareVertexColorBuffer.numItems = 24;
```

Listing 18. Vertices and colors for the cube.

As there are now more items in the buffers the numItems attributes of all the buffers have to be updated as well (12 items for pyramid and 24 for cube).

This addition of vertices and colors is enough for drawing the pyramid but for the cube we will show an alternative method. Instead of using gl.TRIANGLE_STRIP and drawArrays we will be using drawElements function with gl.TRIANGLES. This provides better performance as more vertices can be re-used when displaying 3D objects. For using drawElements we need to create a global parameter called squareVertexIndexBuffer and define it as an element array buffer as shown in Listing 19.

```
squareVertexIndexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, squareVertexIndexBuffer);
var squareVertexIndices = [
      0, 1, 2,      0, 2, 3,    // Front face
      4, 5, 6,      4, 6, 7,    // Back face
      8, 9, 10,     8, 10, 11,  // Top face
      12, 13, 14,   12, 14, 15, // Bottom face
      16, 17, 18,   16, 18, 19, // Right face
      20, 21, 22,   20, 22, 23  // Left face
];
```

```
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(squareVertexIndices),
              gl.STATIC_DRAW);
squareVertexIndexBuffer.itemSize = 1;
squareVertexIndexBuffer.numItems = 36;
```

Listing 19. Element array buffer for the cube.

The numbers in the element array buffer are indices into the vertex position and color buffers. For example the first line means that we will draw a triangle using vertices 0, 1, and 2, and then another using 0, 2 and 3. Because both triangles are the same color and they are adjacent, the result is a square using vertices 0, 1, 2 and 3.

The final change we need to make is to change the drawArrays call for the cube in the drawScene() function into a drawElements call. First we need to bind the element array buffer, then call setMatrixUniforms() and then call the drawElements as shown in Listing 20.

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, squareVertexIndexBuffer);
setMatrixUniforms();
gl.drawElements(gl.TRIANGLES, squareVertexIndexBuffer.numItems, gl.UNSIGNED_SHORT, 0);
```

Listing 20. Changes needed in drawScene.

The resulting WebGL application shows a pyramid rotating around one axis and the cube rotating around another axis as seen in Figure 5.

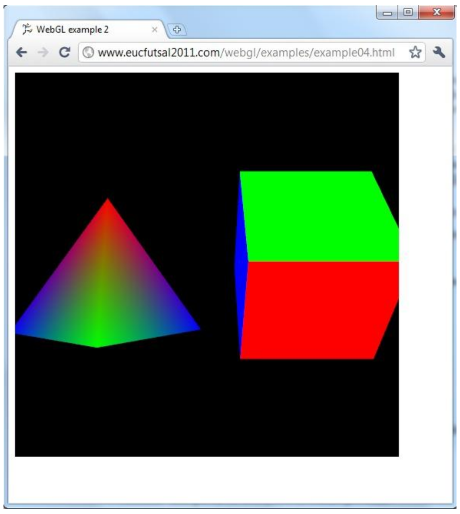

Figure 5: Rotating 3D objects.

3.1.5 Loading textures

In this example we will cover both objects with a texture loaded from a separate file. What is needed for this is to add a function that loads the texture from the file into

WebGL context, add support for textures in shaders, replace objects' color buffers with texture buffers and change the drawScene() code to use the texture for drawing.

Before we start it is good to point out that there has been problems with non-power-of-two (NPOT) textures In WebGL. To avoid the problems it is a good practice to limit the textures with power of 2 dimensions (512x512, 1024x1024 etc.).

The code for loading textures is placed in initTextures() function and can be seen in Listing 21. Also a call for this function needs to be added to the webGLStart function so that the textures are initialized when the web page is loaded.

```
var neheTexture;
function initTexture() {
  neheTexture = gl.createTexture();
  neheTexture.image = new Image();
  neheTexture.image.onload = function() {
    handleLoadedTexture(neheTexture)
  }

  neheTexture.image.src = "nehe.gif";
}

function handleLoadedTexture(texture) {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, texture.image);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
  gl.bindTexture(gl.TEXTURE_2D, null);
}
```

Listing 21. Functions needed for loading a texture.

A JavaScript object called neheTexture is created for which a WebGL reference to a texture is created (gl.createTexture). After that a new JavaScript image is created for the object and texture from nehe.gif is loaded on it with handleLoadedTexture callback function.

The handleLoadedTexture function creates a WebGL texture from the JavaScript image. First it binds the current texture to be the one used and then the texture is flipped vertically to match with the used coordinate system. The gl.texImage2D function call creates the texture from the JavaScript image after which two parameters are passed to effect on how the image is scaled up or down depending how big it is on the screen. The last line unbinds the texture. After this the texture is ready to be used and can be used through the neheTexture global parameter.

Next we will change our shaders to accept textures. The new shader codes can be seen in Listing 22.

```
<script id="shader-fs" type="x-shader/x-fragment">
  #ifdef GL_ES
  precision highp float;
  #endif

  varying vec2 vTextureCoord;

  uniform sampler2D uSampler;

  void main(void) {
    gl_FragColor = texture2D(uSampler, vec2(vTextureCoord.s, vTextureCoord.t));
  }
</script>

<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec2 aTextureCoord;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  varying vec2 vTextureCoord;

  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vTextureCoord = aTextureCoord;
  }
</script>
```

Listing 22. Shaders for accepting textures.

The shader codes should look really similar to the previous ones. This time there are attributes and varyings made for texture coordinates in the same way as the colors were. The only bigger difference can be seen in the fragment shader, where a uniform of the type of sampler2D is created for handling the texture. The actual color of the pixel is taken directly from the texture by texture2D function.

The same modifications need to be done in the initShaders function. Instead of creating a vertexColorAttribute a textureCoordAttribute should be created as in Listing 23. Also the samplerUniform needs to be added to the shaderProgram.

```
function initShaders() {
  var fragmentShader = getShader(gl, "shader-fs");
  var vertexShader = getShader(gl, "shader-vs");

  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Could not initialise shaders");
  }

  gl.useProgram(shaderProgram);
  shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
                                                               "aVertexPosition");
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

  shaderProgram.textureCoordAttribute = gl.getAttribLocation(shaderProgram,
                                                             "aTextureCoord");
  gl.enableVertexAttribArray(shaderProgram.textureCoordAttribute);

  shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
  shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix");
}
```

Listing 23. initShaders function for textures.

Next we will replace the objects color buffers with texture buffers as seen in Listing 24.

```
squareVertexTextureCoordBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexTextureCoordBuffer);
var textureCoords = [
      // Front face
      0.0, 0.0,
      1.0, 0.0,
      1.0, 1.0,
      0.0, 1.0,

      // Back face
      1.0, 0.0,
      1.0, 1.0,
      0.0, 1.0,
      0.0, 0.0,

      // Top face
      0.0, 1.0,
      0.0, 0.0,
      1.0, 0.0,
      1.0, 1.0,

      // Bottom face
      1.0, 1.0,
      0.0, 1.0,
      0.0, 0.0,
      1.0, 0.0,

      // Right face
      1.0, 0.0,
      1.0, 1.0,
      0.0, 1.0,
      0.0, 0.0,

      // Left face
      0.0, 0.0,
      1.0, 0.0,
      1.0, 1.0,
      0.0, 1.0,
];
```

```
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords), gl.STATIC_DRAW);
squareVertexTextureCoordBuffer.itemSize = 2;
squareVertexTextureCoordBuffer.numItems = 24;

triangleVertexTextureCoordBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexTextureCoordBuffer);
var textureCoords = [
        // Front face
        1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0,
        // Right face
        1.0, 0.0, 0.0,
        0.0, 0.0, 1.0,
        0.0, 1.0, 0.0,
        // Back face
        1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0,
        // Left face
        1.0, 0.0, 0.0,
        0.0, 0.0, 1.0,
        0.0, 1.0, 0.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords), gl.STATIC_DRAW);
triangleVertexTextureCoordBuffer.itemSize = 3;
triangleVertexTextureCoordBuffer.numItems = 12;
```

Listing 24. TextureCoordBuffers for objects.

Finally the drawScene function needs to be updated to draw with textures instead of colors. This is done by binding the texture instead of binding the color buffers for both objects. In both cases the color buffer bindings can be replaced with code shown in Listing 25.

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, neheTexture);
gl.uniform1i(shaderProgram.samplerUniform, 0);
```

Listing 25. Binding texture in drawScene.

The resulting WebGL application is shown in Figure 6.

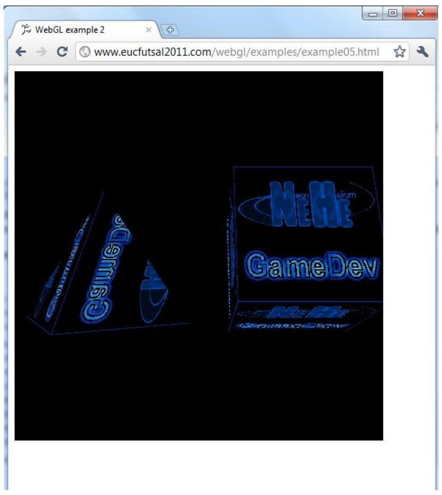

Figure 6: 3D objects with textures.

3.2 Using GLGE for WebGL development

GLGE[18] is one of the most promising WebGL libraries. The library is created and maintained by a single developer, Paul Brunt. Source codes for the library can be found in GLGE Gitorious repository[19]. On the project website the author states that “the aim of GLGE is to mask the involved nature of WebGL from the web developer, who can then spend his/her time creating richer content for the web”.

Main features of the GLGE are (on the website):

1. Keyframe animation
2. Perpixel lighting directional lights, spot lights and point lights
3. Normal mapping
4. Animated materials
5. Skeletal animation (Work In Progress)
6. Collada format support
7. Parallax Mapping
8. Text rendering (probably bitmap)
9. Fog
10. Depth Shadows
11. Shader-based picking
12. Environment mapping
13. Reflections/Refractions
14. Collada Animations
15. Portals (via jiglibJS)

[18] http://www.glge.org
[19] https://github.com/supereggbert/GLGE

Planned additions (pending on technical walls), not necessarily in this order:

1. Shape keys
2. LOD (Level of detail)
3. Culling
4. 2D Filters
5. Primitives Creation

3.2.1 GLGE example

GLGE projects are structured as shown in Figure 7. The application consists of three parts: an HTML file that works as a “boot loader”, a JavaScript file that holds the application logic and an XML file that contains static resources.

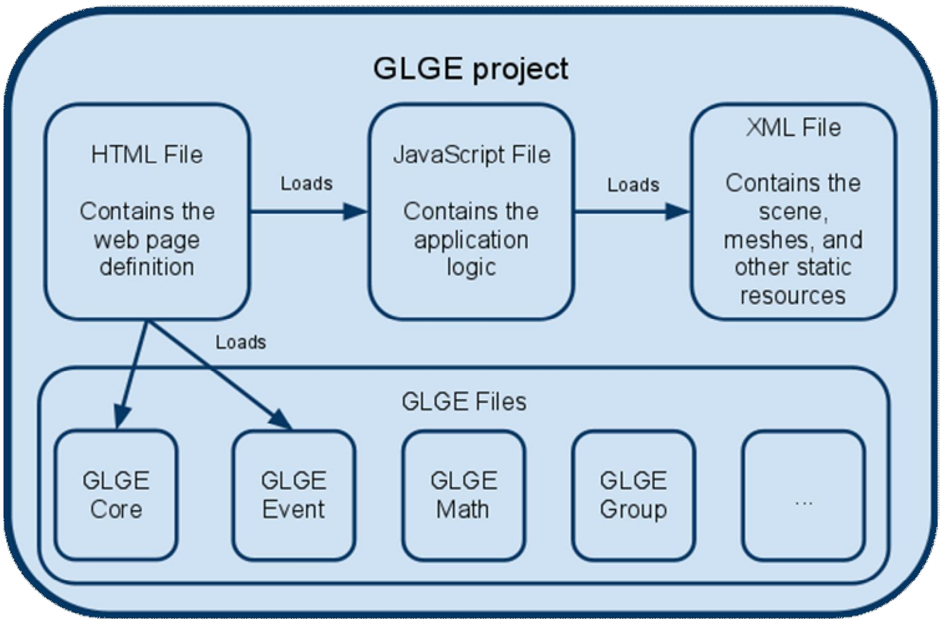
GLGE project
HTML File
Contains the web page definition
Loads
JavaScript File
Contains the application logic
Loads
XML File
Contains the scene, meshes, and other static resources
Loads
GLGE Files
GLGE Core
GLGE Event
GLGE Math
GLGE Group
...

Figure 7: GLGE project structure.

HTML File. The HTML file is needed to define the canvas element where WebGL content can be rendered. In addition it is used to load necessary JavaScript files, such as GLGE components and the application logic. Listing 26 shows how these files are referenced in the HTML markup.

```
<script type="text/javascript" src="glge/glge_math.js"></script>
<script type="text/javascript" src="glge/glge.js"></script>
<canvas id="canvas" width="400" height="400"></canvas>
<script type="text/javascript" src="application_logic.js">
```

Listing 26. The HTML file loads necessary JavaScript resources and defines the WebGL canvas element.

JavaScript File. The application logic is defined in a JavaScript file. It loads the XML file with additional static resources and determines how mouse and keyboard events are handled. 3D objects and textures can be created dynamically in JavaScript.

```
var doc = new GLGE.Document();
doc.onLoad = function() {
  var renderer = new GLGE.Renderer(document.getElementById("canvas"));
  var scene = doc.getElement("mainscene");
  renderer.setScene(scene);

  function render() {
    renderer.render();
  }

  setInterval(render, 15);
}
doc.load("scene1.xml");
```

Listing 27. JavaScript file loads the XML file and runs the render function in every 15 milliseconds.

XML File. The XML file contains static resources for GLGE project. It holds meshes (objects), camera, scene, animations, materials, Collada objects etc. GLGE meshes can be created using an editor, Blender for example. Listing 28 shows an example of XML creating a scene with one box shaped object. Lines containing data for positions, normals, uv1 and faces are generated directly from Blender and are cropped in the Listing 28 because they contain massive amount of numerical data. In addition, camera, light and box material are defined.

```
<?xml version="1.0" ?>
<glge>

  <mesh id="box">
    <positions>1.000000,0.999999,1.000000,...</positions>
    <normals>0.000000,1.000000,0.000000,...</normals>
    <uv1>0.333333,0.498471,0.001020,...</uv1>
    <faces>0,1,2,...</faces>
  </mesh>

  <camera id="maincamera" loc_z="20" />
  <material id="boxmaterial" color="#900" />
```

```
    <scene id="mainscene" camera="#maincamera" ambient_color="#fff">
      <light id="mainlight" loc_y="5" type="L_POINT" />
      <object id="box" mesh="#box"
        rot_x="-.8" rot_y=".5"
        material="#boxmaterial" />
    </scene>

  </glge>
```

Listing 28. Example of a GLGE XML file.

GLGE Files. GLGE is divided into modules, and only necessary parts can be loaded into the application. The modules and their purposes are:

- Core (The main GLGE functionality)
- Animatable (Animation framework)
- Document (Class to load scene, object, mesh etc from an external XML file)
- Event (Event handling)
- Group (Class to allow object transform hierarchies)
- JSON Loader (Class to load objects in JSON format)
- Math (Class to perform mathematical operations)
- Messages (GLGE Messaging System)
- Placeable (Abstract class to argument objects that requires position, rotation and scale)
- Quicknotation (class to implelement quick notation)

The outcome of this GLGE example can be seen in Figure 8.

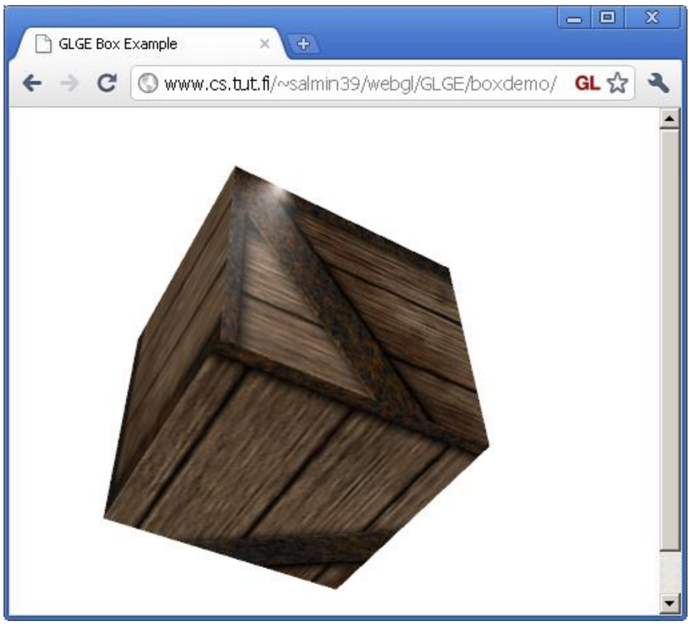

Figure 8: GLGE example in Chrome.

3.2.2 Notes about using GLGE

By comparing amount of lines needed to display textured WebGL objects, GLGE successfully hides many WebGL programming details. If meshes and textures can be created with an editor, the application development is relatively straightforward. However, defining the scene (camera, lights and object positions) by hand may be difficult for some programmers. In general GLGE seems to be a bit biased towards 3D games and worlds (e.g. 3D galleries).

GLGE only offers brief online documentation of all the classes but it lacks more detailed explanations. It is not clear how some of the functionalities should be used if an example is not available. There are no instructions on how the XML file should be formatted and what kind of parameters different elements can have. Examples are available, though. Creating objects with JavaScript is even more difficult as there are no examples about using the framework this way. Currently an application developer needs to be able to read and understand GLGE sources to be able to use some of the functionalities.

4. FURTHER READING AND THINGS TO KNOW ABOUT

A very good site to explore for more advanced WebGL examples is http://learningwebgl.com/ blog which have tutorials covering topics such as different kinds of lighting, loading a JSON model, handling mouse events and key inputs and how to render a WebGL scene itself into a texture. Also WebGL framework C3DL's tutorials give insight on various aspects of 3D graphics in general.

Other good sites to visit for more WebGL examples are Khronos web site[20] and especially its demo repository[21] and User Contributions page[22]. Also the Chrome Experiments page[23] and Mozilla's Web'O Wonder[24] includes various WebGL experiments.

Khronos group have just started a WebCL Working Group to explore defining a JavaScript binding to the Khronos OpenCL™ standard for heterogeneous parallel computing. WebCL creates the potential to harness GPU and multi-core CPU parallel processing from a web browser, enabling significant acceleration of applications such as image and video processing and advanced physics for WebGL games.

Another way to improve JavaScript performance is to optimize the math calculations. If the Sylvester.js matrix library seems too slow there are other more optimized libraries available as well. One actively maintained one and fast is glMatrix[25]. Another notable utility library for WebGL is called webgl-utils.js which is provided by Google[26]. One of the neatest features it offers is the requestAnimFrame cross-browser compatibility function that disables WebGL graphics rendering when the tab is not active. For handling key inputs a JavaScript library called Shortcuts.js[27] might prove helpful. There is also a 3D physics library available for JavaScript called JigLibJS[28] that can be used in WebGL applications as well.

As WebGL is displayed in the web browser it is good to remember all the other possibilities offered by other web technologies. For example, if you want to make your WebGL application to operate in fullscreen mode it can be done with CSS and JavaScript by hooking the document.resize event and detecting the window and screen sizes. After that the <canvas> element can be resized to fullscreen with CSS.

Even though our WebGL examples shown in the previous chapters are using global variables, in most of the cases it is not advisable to do everything that way. It is very handy to use functional programming techniques in JavaScript to write more elegant code [5].

It is also good to point out that it is best to test WebGL code on all browsers and platforms needed. There might be huge performance differences on different platforms and even on different browsers. As WebGL bases on OpenGL ES 2.0 a good enough graph-

[20] http://www.khronos.org/webgl/
[21] http://www.khronos.org/webgl/wiki/Demo_Repository
[22] http://www.khronos.org/webgl/wiki/User_Contributions
[23] http://www.chromeexperiments.com/webgl/
[24] https://demos.mozilla.org/en-US/
[25] http://code.google.com/p/glmatrix
[26] https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/sdk/demos/common/webgl-utils.js
[27] http://www.openjs.com/scripts/events/keyboard_shortcuts
[28] http://www.jiglibjs.org/

ics card might be needed to display WebGL efficiently and correctly. There is also a list available of the blocked graphics drivers that have got problems running WebGL in Firefox[29]. To make it easier to test WebGL applications a web browser add-on called WebGL Inspector[30] is available and worth taking a look.

For further reading on the subject of how HTML5 and WebGL might change the whole software development scene can be found in the publications we have made so far in our Lively Goes 3D project: The Death of Binary Software: End User Software Moves to the Web [8] and Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces [9].

[29] https://wiki.mozilla.org/Blocklisting/Blocked_Graphics_Drivers
[30] http://benvanik.github.com/WebGL-Inspector/

5. CONCLUSIONS

In this technical report we introduced two novel technologies, HTML5 and WebGL, which were found successful for development of visually rich web applications. The gap between desktop and web applications has been decreased notably thanks to these two technologies. Our experiences about HTML5 as well as WebGL are very promising and we see that they have a central role in the way the cutting edge web software is developed in the future.

The WebGL standard recently gained stability as 1.0 version was published. HTML5, however, is still a draft and prone to changes. Furthermore, support for these technologies varies between different browsers. This adds another challenge for a web developer. It is likely, however, that the situation will be better as standardization work continues. Meanwhile, working with somewhat incomplete set of tools is necessity if one wishes to use these technologies.

Different kinds of WebGL frameworks have been developed intensively. New features are added almost weekly to actively developed frameworks. However, in some cases the development has been completely ceased. Quality of the frameworks is varying and choosing one to be used as a base for application other than demo purposes is still very difficult. It is hard to predict which one will provide enough support for serious development efforts in the future. However, some of the frameworks seem promising and have enterprise support behind them, too.

As stated above, the WebGL and HTML5 bring the web one step closer to the desktop as an application platform. The new technologies include many familiar techniques from desktop application development including drag-and-drop, sockets, offline file storages, media elements and OpenGL graphics. The experiments show that adding these features into browser's toolbox, allows even full-fledged 3D games to be ported to a web browser (see for instance Quake 2 implemented in WebGL[31]).

However, web applications do not have to follow the same restrictions as their desktop counterparts. Web opens up a whole new market for applications that include collaborative and social features in unforeseen scale. In addition, combining data from various web pages will make it possible to build up 3D mashups for various different use cases including advanced visualization of complex data, multiuser virtual worlds, different product catalogs, collaborative 3D planning and drawing applications or virtual displays for museums.

[31] http://playwebgl.com/games/quake-2-webgl

REFERENCES

[1] World Wide Web Consortium. HTML 5.0 Specification. Technical Specification, June 24, 2010.

[2] Khronos Group. WebGL Specification, Version 1.0. Technical Specification, February 10, 2011. Available online: https://www.khronos.org/registry/webgl/specs/1.0/ Retrieved March 6, 2011.

[3] Kessenich, J., Baldwin, D., Rost, R., *The OpenGL Shading Language*. Technical Specification,, July 24, 2010. Available online: http://www.opengl.org/registry/doc/GLSLangSpec.4.10.6.clean.pdf Retrieved June 20, 2011.

[4] Anttonen, M. Lively Goes 3D Report – Task 1: Analysis. Internal Document, March 30, 2010.

[5] Crockford, D. JavaScript: The Good Parts. Publisher: O'Reilly Media / Yahoo Press. Released May 2008.

[6] Hearn, D., Baker, M. Computer Graphics with OpenGL. Publisher: Prentice Hall. ISBN-10: 0130153907, ISBN-13: 9780130153906.

[7] Khronos Group. The OpenGL ES Shading Language, Version 1.00. Technical Specification, May 12, 2009. Available online: http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf Retrieved March 6, 2011.

[8] Taivalsaari, A., Mikkonen, T., Anttonen, M., Salminen, A., *The Death of Binary Software: End User Software Moves to the Web*. In Proceedings of the 9th International Conference on Creating, Connecting and Collaborating through Computing (C5-2011, Kyoto, Japan, 18-20 January, 2011).

[9] Anttonen, M., Salminen, A., Mikkonen, T., Taivalsaari, A., *Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces*. To appear in Proceedings of the 26th ACM Symposium on Applied Computing (SAC'2011, TaiChung, Taiwan, March 21-25, 2011).

Attachment 1 – Source code for the first WebGL example

```
<!DOCTYPE html>
<html>
<head>
<title>WebGL example 1</title>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
<script type="text/javascript" src="sylvester.js"></script>
<script type="text/javascript" src="glUtils.js"></script>

<script id="shader-fs" type="x-shader/x-fragment">
    #ifdef GL_ES
    precision highp float;
    #endif
    void main(void) {
        gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
</script>

<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;

  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
  }
</script>

<script type="text/javascript">

  var gl;
  function initGL(canvas) {
    try {
      gl = canvas.getContext("experimental-webgl");
      gl.viewport(0, 0, canvas.width, canvas.height);
    } catch(e) {
    }
    if (!gl) {
      alert("Could not initialise WebGL, sorry :-(");
    }
  }


  function getShader(gl, id) {
    var shaderScript = document.getElementById(id);
    if (!shaderScript) {
      return null;
    }

    var str = "";
    var k = shaderScript.firstChild;
    while (k) {
      if (k.nodeType == 3) {
        str += k.textContent;
      }
      k = k.nextSibling;
    }
    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
      shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
      shader = gl.createShader(gl.VERTEX_SHADER);
    } else {
      return null;
    }

    gl.shaderSource(shader, str);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert(gl.getShaderInfoLog(shader));
      return null;
    }

    return shader;
  }


  var shaderProgram;
  function initShaders() {
    var fragmentShader = getShader(gl, "shader-fs");
    var vertexShader = getShader(gl, "shader-vs");

    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert("Could not initialise shaders");
    }

    gl.useProgram(shaderProgram);

    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,"aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

    shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
    shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix");
  }


  var mvMatrix;
  function loadIdentity() {
    mvMatrix = Matrix.I(4);
  }


  function multMatrix(m) {
    mvMatrix = mvMatrix.x(m);
  }


  function mvTranslate(v) {
    var m = Matrix.Translation($V([v[0], v[1], v[2]])).ensure4x4();
    multMatrix(m);
  }


  var pMatrix;
  function perspective(fovy, aspect, znear, zfar) {
    pMatrix = makePerspective(fovy, aspect, znear, zfar);
  }
```

```
function setMatrixUniforms() {
  gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, newFloat32Array(pMatrix.flatten()));
  gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, new Float32Array(mvMatrix.flatten()));
}

var triangleVertexPositionBuffer;
var squareVertexPositionBuffer;
function initBuffers() {
  triangleVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
  var vertices = [
       0.0,  1.0,  0.0,
      -1.0, -1.0,  0.0,
       1.0, -1.0,  0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  triangleVertexPositionBuffer.itemSize = 3;
  triangleVertexPositionBuffer.numItems = 3;

  squareVertexPositionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
  vertices = [
       1.0,  1.0,  0.0,
      -1.0,  1.0,  0.0,
       1.0, -1.0,  0.0,
      -1.0, -1.0,  0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  squareVertexPositionBuffer.itemSize = 3;
  squareVertexPositionBuffer.numItems = 4;
}

function drawScene() {
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  perspective(45, 1.0, 0.1, 100.0);
  loadIdentity();

  mvTranslate([-1.5, 0.0, -7.0]);

  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, triangleVertexPositionBuffer.itemSize,
                         gl.FLOAT, false, 0, 0);
  setMatrixUniforms();
  gl.drawArrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.numItems);


  mvTranslate([3.0, 0.0, 0.0]);
  gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, squareVertexPositionBuffer.itemSize,
                         gl.FLOAT, false, 0, 0);
  setMatrixUniforms();
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, squareVertexPositionBuffer.numItems);
}

function webGLStart() {
  var canvas = document.getElementById("example-canvas");
  initGL(canvas);
  initShaders();
  initBuffers();

  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.enable(gl.DEPTH_TEST);

  drawScene();
}
```

```html

</script>

</head>
<body onload="webGLStart();">
  <canvas id="example-canvas" style="border: none;" width="500" height="500"></canvas>
</body>
</html>
```

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland