International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:7, No:2, 2013

A Standalone WebGL Supporting Architecture

Nakhoon Baek

***Abstract*—**WebGL is typically used with web browsers. In this paper, we represent a standalone WebGL execution environment, where the original WebGL source codes show the same result to those of WebGL-capable web browsers. This standalone environment enables us to run WebGL programs without web browsers and/or internet connections. Our implementation shows the same rendering results with typical web browser outputs. This standalone environment is suitable for low-tier devices and/or debugging purposes.

***Keywords*—**WebGL, OpenGL ES, stand-alone, architecture

I. Introduction

WEBGL is the standardized way of providing 3D graphics output on web browsers, and first released in March 2011 [1]. More precisely, it is based on the canvas elements from HTML5 document architecture, and provides full 3D graphics features of OpenGL ES 2.0 and OpenGL ES SL (shader language) specifications. As shown in Fig. 1, WebGL programs are written in JavaScript language and embedded into HTML5 documents. The JavaScirpt codes call the WebGL API functions, to finally get 3D output on the web browser, using the underlying OpenGL ES 2.0 or DirectX 9.0 hardware.

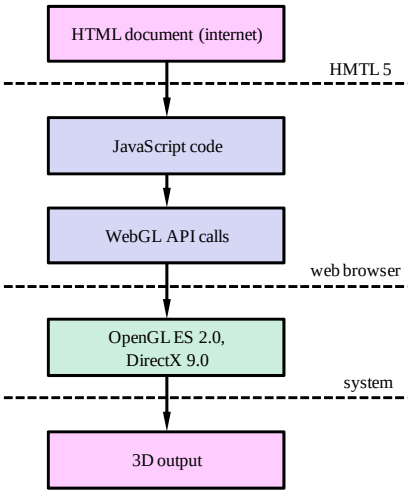


Fig. 1 Overall flow of WebGL-based 3D graphics output

WebGL achieved full compatibility and independence with underlying operating systems and/or windowing systems, adopting web browsers as the contents providing platform. Notice that all the previous OpenGL family specifications are not free from the underlying platforms. Now, web browsers can provide overall programming interfaces, which are independent of underlying hardware, middleware, graphics library, and so on. Ideally, a WebGL program can be executed on every platform, where a web browser is available.

A web browser-based graphics environment, such as WebGL, would be platform-independent and hardware-independent. Additionally, it requires zero-footprint, since the web browsers are based on the internet connections, and can provide 3D graphics contents as streaming services. In contrast, we cannot use the WebGL-based contents, without web browsers or internet connections. Though the web browser-based environment was used to solve many compatibility-related problems, it is now impossible to use WebGL alone, without web browsers or internet connections.

In this paper, we present a standalone WebGL-executable environment. Where JavaScript programs with WebGL API calls can produce 3D graphics rendering results. This kind of standalone environment allows execution of WebGL programs without any web browsers and/or internet connects, to finally achieves more consistency of user experiences. It can be also used for local development and debugging purposes, as a light-weight programming environment.

Some related works are presented in Section II. Design and implementation details of our system are followed in section III. in section IV, Rendering results are compared with those of existing WebGL-capable web browsers. Section V presents our conclusions and future work.

II. Previous Works

WebGL was originally introduced to add full 3D graphics rendering features to the HTML5canvas element. Thus, previous WebGL-capable systems are all implemented on the HTML5-capable web browsers. Currently, most major web browsers, including Google Chrome, Mozilla Firefox, Opera Opera, and Apple Safari, provide WebGL features on their desktop versions, and are ready for providing them on their mobile versions. In the case of Microsoft Internet Explorer, they have not announced any official plane for WebGL support, but some third-party companies already provide special plug-in programs for full WebGL support [2].

Most of old pre-HTML5 web browsers do not provide WebGL features. Some of lightweight web browsers for mobile devices are also impossible to support full WebGL features. For these cases, a full software WebGL emulation library [3] has been implemented in JavaScript language. Since this library is executed in the JavaScript interpreter, it is hard to be used for full-scale animations or interactive applications.

Angle project is one of interesting WebGL related

Prof. Nakhoon Baek is with the School of Computer Science and Technology, Kyungpook National University, Daegu 702-701, Korea (e-mail: oceancru@gmail.com).

This investigation was financially supported by Semiconductor Industry Collaborative Project between Kyungpook National University and Samsung Electronics Co. Ltd.

implementations [4]. In Microsoft Windows environment, they use DirectX as the official 3D graphics library, while WebGL implicitly requires OpenGL 2.0 or its compatible library. Angle project aims to process WebGL function calls directly on the DirectX 9.0 library, to let the WebGL functions be executable without OpenGL related drivers. It is now under development, and expected to be used for future applications.

There have been continuous attempts to execute the same web contents on both of the web browsers and other standalone application programs. For example, to execute JavaScript programs in separate application programs, the *SpiderMonkey* JavaScript Engine [5] was extracted from Mozilla Firefox browser, and V8 JavaScript engine [6] from Google Chrome. With these JavaScript Engines, JavaScript codes executed on web browsers can be embedded into traditional procedural languages, such as C and C++, to finally show the same execution results. One of the SpiderMonkey-based application programs [7] even provide standalone JavaScript programming environment. Notice that these applications are devoted to the execution of pure JavaScript language, and do not provide other extra features such as WebGL.

In the case of WebGL, the official standard specification was released in March 2011, and thus, most previous efforts are concentrated on the conversion process of previous desktop OpenGL or OpenGL ES applications to WebGL-based ones. At least to our best knowledge, there was no attempt to execute WebGL programs directly on separate application programs without web browser support.

III. Overall Architecture

In this paper, we present a stand-alone hardware-accelerated application program executing WebGL programs written in the JavaScript language. Our light-weight standalone application program will accept HTML5 documents or their embedded JavaScript programs with WebGL API calls, and directly display their final 3D graphics results on the screen, rather than with web-browsers. Using hardware OpenGL features provided on desktop PC's and smart-phones, our system will be fully hardware accelerated.

The overall architecture of our system is shown in Figure 2. Starting from parsing the JavaScript source codes, our system converts their WebGL API calls to suitable OpenGL or OpenGL ES 2.0 functions. Finally, the underlying native OpenGL hardware shows the rendering result on the screen.

To be executed on the standalone program rather than web browsers, we applied some restrictions on the HTML5 and JavaScript-based WebGL programs as follows:

- *Multi-canvases are not supported.* – Our standalone system processes a single WebGL program, while web-browsers can handle multiple canvases in a HTML5 document.
- *Text outputs from the JavaScript programs or HTML5 documents are completely ignored.* – Pure OpenGL systems are hard to process the various text features of the HTML5 standard. We will mainly focus on the 3D graphics output from the WebGL.
- *Only selected features are accepted from the HTML5 DOM architecture.* – The DOM (document object model) interface [8] is the standard way of fetching information from the HTML documents, and provides variety of features. In contrast, WebGL programs only use several typical way of information access. Thus, we only provide some selected features of the DOM interface. For example, a typical WebGL program requires vertex shader and fragment shader source codes, which are typically provided in separate nodes in the DOM architecture. We implemented the way of gathering shader source codes from the different nodes in the DOM architecture. However, we do not provide all the general ways of gathering any type of information from the nodes.
- *Selected options from the canvas tag are accepted.* – WebGL programs are typically displayed with the HTML5 canvas tags. This canvas tag is originally designed to show

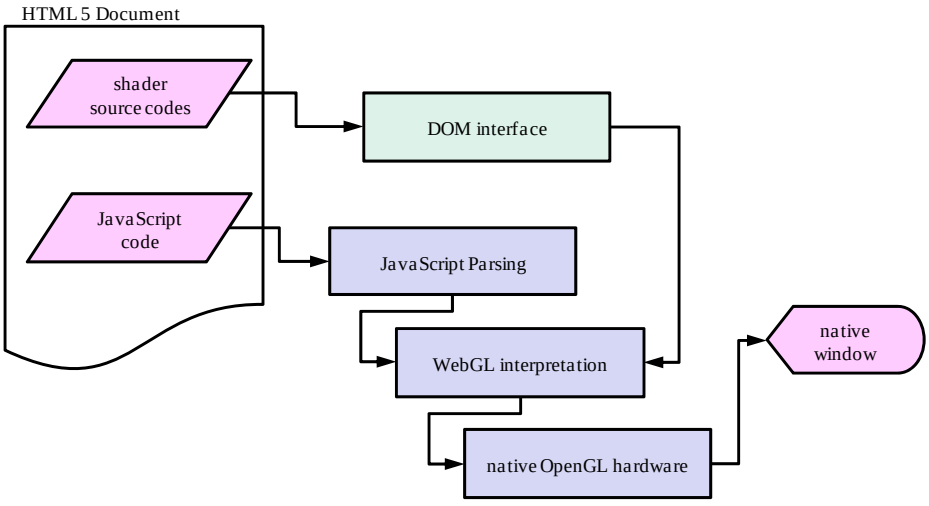


Fig. 2 The overall architecture of our standalone WebGL system

various 2D and 3D graphics output, with plenty of output options. Among them, our system only accepts several options directly related to the WebGL output.

From the early stage of our system architecture design, we focused on the lightweight standalone environment, to provide WebGL features. In the development stage, we aimed to provide prototype features as soon as possible, with minimum amount of effort, through fully utilizing existing implementations. Thus, our overall standalone environment is composed of the following elements:

- *Small DOM emulator* – Only selected DOM features are implemented.
- *Customized SpiderMonkey JavaScritp Engine* – We naturally need a JavaScript engine to parse JavaScript programs. We selected the SpiderMonky JavaScript Engine, originally used for the Mozilla Firefox web browser. In this SpiderMonkey Engine, programmers can provide extra C/C++ functions, for the features required by the JavaScript program, for example, such as WebGL API calls. We actually provide C functions performing required steps for each WebGL API calls.
- *Customized codes to use OpenGL and GL SL features* – WebGL API calls are processed with underlying hardware OpenGL and GL Shader Language features. Notice that there are a lot of differences in the details of WebGL and OpenGL specifications. For example, OpenGL drivers cannot understand all WebGL-specific parametric values, and we need extra processing for each of these exceptional cases. We carefully checked all of these non-compatible cases, and provided suitable modifications to finally execute WebGL programs on the OpenGL and GL SL environment. Actually, most of our work was focused on these customization process.
- *User interaction based on the GLUT library* – In HTML5, DOM objects and canvas tag provide user interaction features, including keyboard input, mouse control, and animation loops with pre-specified time intervals. We realized these user interaction features with keyboard, mouse, and idle callback functions in the GLUT library.
- *Image loading libraries for texture support* – In the DOM architecture, user specifies the texture image filename, and the system automatically load the texture file through internet connections. In contrast, our major components, JavaScript engine, OpenGL, GL Shader Language, and GLUT library do not provide any texture loading features. Thus, we integrated image loading functions, for all image file formats available in the DOM architecture.

After a set of integration and fine tuning steps, we get the prototype standalone WebGL system.

IV. Implementation Results

Our implementation works as an independent program on Windows7 PC's, with OpenGL and GLUT supports. Core features and shader language-based features of the WebGL specification work successfully, through combining our customized codes, the JavaScript engine, underlying OpenGL and native Window systems. As shown in Figure 3, our execution results are same to those of WebGL-enabled browsers, for variety of test programs. During these test procedures, we found several technical issues, and solved them, as shown as follows:

Flipping texture images: OpenGL specification traditionally assumes that the positive *y* axis is upward, while web browsers use downward *y* axes. In WebGL specification, PixelStore(…) function may be used to set UNPACK_FLIP_Y flag, which actually flips the y axis of the given texture images. Since underlying OpenGL does not yet support this feature, we flip the texture image, if necessary, at its loading time.

Supporting animation loops: In its animation sequences, HTML5 accomplished next frames through calling its requestAnimFrame(…) function. Its time intervals are not specified in real HTML5 documents. In our case, we achieved smooth animations through generating animation ticks for every 40 msec.

Supporting keyboard interactions: WebGL-enabled web browsers pass the user's key press and key release actions directly to WebGL programs. Our stand-alone implementation also does it. Our implementation passes all the key press and key release events even for special keys including *alt*, *shift* and function keys, based on the GLUT features.

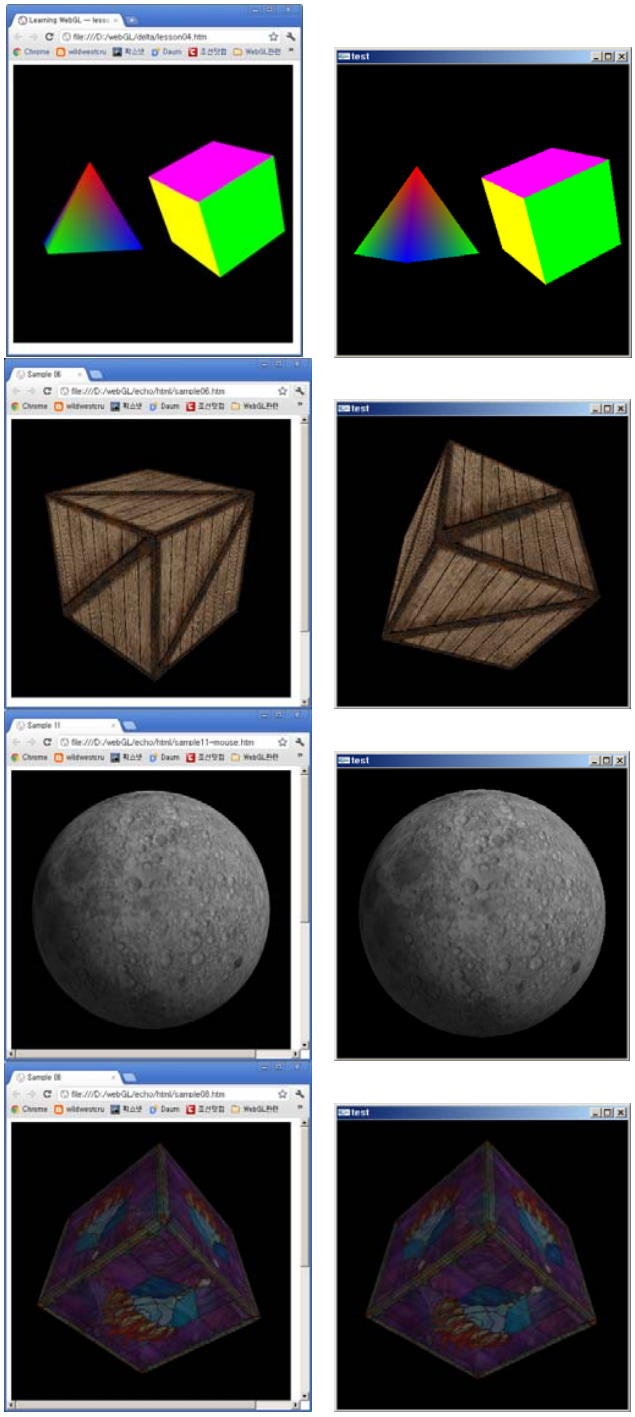
Supporting mouse interactions: Mouse motions and mouse button events are also passed to the WebGL programs. Using GLUT features, we also pass all the mouse motion and button events to the program.

Our current implementation shows the same graphics output to the WebGL-enabled web browsers, as shown in Figure 3. Additionally, it shows the same response for keyboard and mouse interactions. In its execution speed, our program works slightly faster, mainly due to its lack of network delays. Since it is hard to measure the execution speed of WebGL programs, we cannot compare the execution speeds. However, we found that there have been no remarkable speed differences.

V. Conclusion

In this paper, we presented a standalone 3D graphics programming environment, which shows the rendering results from WebGL-based JavaScript programs, without web browser supports. Although the WebGL was originally designed for web browser-based, platform-independent 3D graphics output, we still need its standalone, local programming environment. In this paper, we starts from the SpiderMonkey JavaScript engine, OpenGL, GL shader Language, and GLUT libraries, and added much customized codes, to finally get the standalone WebGL execution environment.

We presented the details of our system design and implementation. We plan to add some related features and some official WebGL extensions.



(a) (b)

Fig. 3 Rendering results (a) from Chrome web browser and (b) our result

ACKNOWLEDGMENT

This investigation was financially supported by Semiconductor Industry Collaborative Project between Kyungpook National University and Samsung Electronics Co. Ltd.

REFERENCES

[1] C. Marrin Ed., WebGL Specification, Draft 16, Khronos Group, 16 March 2012.
[2] "IEWebGL: WebGL for Internet Explorer", http://iewebgl.com, on 19 Mar 2012.
[3] C. Shanahan, "cWebGL: WebGL stack in JavaScript", http:cimanron.net/, on 14 Feb 2012.
[4] "ANGLE: Almost Native Graphics Layer Engine", http://code.google.com/angleproject/, on 24 March 2012.
[5] Mozilla Developer Network, SpiderMonkey, http://developer.mozilla.org/en/SpiderMonkey, on 12 Dec 2011.
[6] Google, "V8 JavaScript Engine", http://code.google.com/p/V8/, on Dec 2011.
[7] "jslibs: standalone JavaScript development runtime environment", http://jslibs. googlecode.com, on 29 Feb 2012.
[8] W3C, Document Object Model (DOM) Level 3 Core Specification, Version 1.0, W3C Recommendation, 2004.

Nakhoon Baek is currently an associate professor in the School of Computer Science and Engineering at Kyungpook National University, Korea. He received his B.A., M.S., and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1990, 1992, and 1997, respectively. His research interests include graphics standards, graphics algorithms and real-time rendering. He is now also the Chief Engineer of Mobile Graphics Inc., Korea.