2012

Networked virtual environments with Javascript, WebSockets and WebGL

McGregor, Don

Networked Virtual Environments With Javascript, WebSockets and WebGL

Don McGregor
Don Brutzman
Naval Postgraduate School, MOVES Institute
700 Dyer Road, Bldg 246, Rm 265
Monterey, CA 93943
mcgredo@nps.edu
brutzman@nps.edu

Abstract: *A mix of new technologies can enable networked 3D virtual environments in web browsers: WebSockets, Javascript-based DIS, and WebGL and 3D graphics frameworks that rest on top of WebGL. WebSockets is a joint W3C / IETF draft standard that provides TCP connections from the browser to the server without the overhead or latency of HTTP encapsulation or HTTP polling. The DIS standard is used with Javascript to provide bidirectional standards-based entity updates between the browser and the server. WebGL and supporting graphics framework scene graphs provide hardware-accelerated 3D graphics with Javascript bindings in supporting browsers. Together, these technologies allow standards-compliant virtual environments to be deployed inside the browser without plugins. This work reports experimental results from piping DIS streams to WebGL-based frameworks in the browser to create Networked Virtual Environments. The broad availability of DIS streams in Web browsers and mobile devices provides significant new opportunities for the Modeling and Simulation community.*

Background. Networked Virtual Environments (NVEs) have been implemented inside of web browser pages for some time. The idea has considerable appeal since almost all users have a web browser installed on their desktop or mobile device, and NVE content can be kept on the server side where it is easy to update and distribute. Some of the first web-based NVE implementations used web browser plugins such as VRML for 3D graphics and network access [1, 2]. However, the requirement that a plugin be installed often caused problems for users that had no control over the configuration of their desktops. Users often faced a long and bureaucratic process to get a plugin approved before they could view the NVE. The networking portion of the plugins frequently used network ports not approved by standard enterprise firewall rules, resulting in a separate, even more difficult approval process. This made the user experience far from ideal; instead of simply clicking on a web browser link to participate in a NVE users had to engage in considerable bureaucratic negotiations.

Recent technology and standards work has renewed the promise of at least small to medium scale NVEs in web browsers with reduced or non-existent organizational friction. The WebGL standard [3] is an implementation of OpenGL with Javascript bindings. This allows the use of hardware accelerated 3D graphics inside the web page from the Javascript language. WebGL is implemented inside the browser by the browser vendor and eliminates the need for a plugin installation. It has been implemented in various releases of Firefox, Chrome, Safari, and Opera desktop browsers and some mobile browsers. Microsoft uses an equivalent but alternative technology, Silverlight, and has not announced support for WebGL. An alternative but proprietary Microsoft-based NVE implementation could be created using many of the techniques discussed in this paper.

Just as in conventional OpenGL graphics programming, an additional software framework can use WebGL to provide higher-level abstractions such as scene graphs and complex 3D models loaded from storage. Several of these frameworks have emerged, including X3DOM [4], GLGE [5], Three.js [6], and Copperlicht [7]. We have initially implemented an NVE in GLGE for reasons of initial programmer convenience, but plan on moving to X3DOM and X3D in order to exploit the large model base available in the X3D environment.

3D graphics are only part of the problem in implementing a NVE; networking is also needed to exchange entity state information. Classically web pages, once downloaded, were static; the browser contacted the server, downloaded content, and then terminated the connection to the server, thus eliminating any possibility of updates. Web pages became more dynamic with

the advent of Asynchronous Javascript and XML (AJAX) programming, which allows the web page to periodically poll the web server and change its Document Object Model (DOM) to reflect new information obtained from the server. AJAX allows considerably more dynamic and interactive web pages, as seen in applications such as Google Maps and many other web applications. However, polling is an inherently high latency operation while NVEs require low latency to accurately reflect the status of entities in the world. AJAX traffic is also passed encapsulated within the HTTP protocol, which leads to considerable bandwidth and processing overhead as the HTTP headers are stripped away to access the payload. On the other hand, traffic is typically passed between the client web browser and the web server over TCP ports 80/443, ports on which enterprise firewalls nearly universally allow clients to establish outgoing connections.

WebSockets are a joint Internet Engineering Task Force (IETF) and World Wide Web Consortium (W3C) standard for implementing bi-directional, full-duplex communications channels over a single TCP connection from the browser to the web server. The IETF is working on WebSockets as RFC-6455 [8] (its current state is "proposed standard") while the W3C has specified the programming language API to be used to access them. As of this writing the W3C standard is in "Last Call Working Draft" status [9]. WebSockets work by sending an initial handshake to the web server using an HTTP-like message requesting that a WebSocket be opened. The server responds and begins allowing the use of the underlying port 80/443 TCP connection for WebSocket traffic. In effect the WebSocket traffic is negotiated as an HTTP TCP socket but bypasses the HTTP semantics and overhead, and is not encapsulated inside of HTTP requests. The traffic over the WebSocket is minimally framed and lacks the application layer HTTP framing of AJAX. WebSockets are supported in many recently released desktop web browsers, including Internet Explorer, Firefox, Chrome, Safari, and Opera, and several mobile browsers.

As with any NVE, using TCP sockets can present its own problems in comparison to using UDP, including increased latency and jitter, and problematic behavior in network environments that require packets to be resent. However many NVEs can be made to work acceptably over TCP.

Javascript is a scripting language that is nearly universally supported in web browsers. It is weakly typed, prototyped-based, and vaguely reminiscent of Scheme and Self, with C-style syntax. The presence of Javascript in the browser allows developers to implement complex logic in NVEs, including physics, Artificial Intelligence, and other computation.

These three technologies together—hardware accelerated 3D in the browser and a low-latency, bidirectional network connection, combined with Javascript for program logic—enable the implementation of a NVE.

Implementation NVEs exchange information about entity state over the network. The format in which the state information is exchanged is an important practical choice. In the case of military NVEs it is useful to choose an existing standard, to the extent possible. In this case we chose to use a variant of the Distributed Interactive Simulation (DIS) format [10]. DIS has a number of virtues, foremost among them simplicity. It also specifies a packet format and packet contents rather than an API, which allows a Javascript implementation to be done with only a small amount of work.

DIS is a binary protocol, but Javascript, a scripting language, is primarily intended to process text. This led us to convert binary format DIS messages to Javascript Object Notation (JSON) format [11] when communicating with the web browser. The semantic content of the JSON format is identical, but rather than being in a binary format it is transformed to a simple text format. This simplifies the handling of DIS inside the web browser by eliminating binary data parsing.

A diagram of the NVE we implemented is show in figure 1:

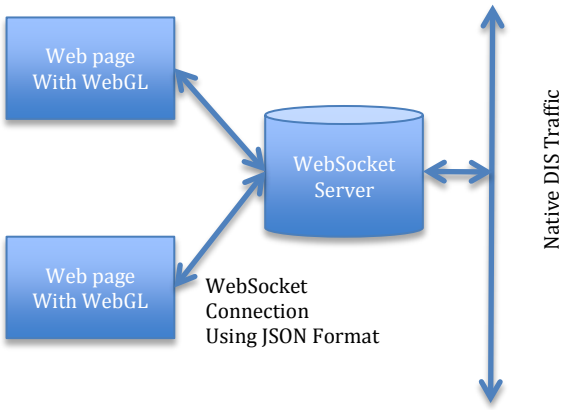

Figure 1: Architecture

Web browsers establish a connection to the server by loading a web page, for example index.html. The HTML file can contain both HTML and Javascript, specifically Javascript that calls WebGL. If using an additional framework to provide scene graph functionality the Javascript necessary to do this is also downloaded. Models, textures, and other graphical content can also be downloaded to the browser. On page load the Javascript establishes a WebSocket connection to the server. This establishes a low-latency, bi-directional, full duplex connection to the server. The web server is in this case running a Java servlet container that contains the code necessary to establish a WebSocket on the server and the server-side communications logic. The servlet sends DIS Entity State PDUs (ESPDUs) to the web page in JSON format, and Javascript within the web page draws entities using WebGL and the information contained in the JSON-format DIS messages. The user can move entities he controls within his web page using the mouse or keyboard. These changes cause JSON format ESPDUs to be sent back to the server, where they can be relayed to other connected clients.

The servlet also listens to native, binary format IEEE-1278.1 DIS traffic on the server's network. This allows it to gateway native DIS traffic to the web clients. Open-DIS [12] was used to read native DIS and translate it to JSON format. Likewise, JSON format DIS messages sent from the web page can be gatewayed into native DIS format on the backbone network. The servlet acts as a communications hub that relays messages among web pages and native DIS traffic on the server's network.

Because all the traffic seen by the web clients is going through a central point at the servlet we can perform other functions as well, including area of interest management or Distributed Data Management (DDM), perform coordinate system transforms on the server side, and so on. For example, a web client concerned with tanks and ground combat could ask the central servlet to not forward to it any ESPDUS from undersea entities. DIS uses a geocentric coordinate system, while many applications use a local, Euclidian coordinate system. If the web page defines a coordinate origin system and tells the servlet of this, we can convert coordinate systems on the server side instead of attempting this in Javascript.

An extract from part of a DIS ESPDU in JSON format is shown in figure 2.

```
{"protocolVersion":6,"exerciseID":0
,"pduType":1,"protocolFamily":1,"ti
mestamp":0,"pduLength":144,"padding
":0,"entityID":{"site":0,"applicati
on":0,"entity":2},"forceId":0,"numb
erOfArticulationParameters":0,"enti
tyType":{"entityKind":1,"domain":1,
"country":225,"category":2,"subcate
gory":5,"spec":0,"extra":0 …
```

Figure 2. Portion of DIS PDU in JSON Format

JSON is a very simple text-based format that directly maps object instance variable names to values. Because Javascript is weakly typed and uses a prototype-based class system, sometimes known as "Duck-Typing" (if it quacks like a duck it's a duck) there is no need to write Javascript DIS classes on the web client side. It is simpler and more straightforward to simply convert a Java DIS object into the JSON format DIS on the server side, send it to the client, and then turn it into a Javascript object via the Javascript eval() function. Conversely, to create an ESPDU on the web browser side one can simply utilize a prototypical ESPDU Javascript object and change the necessary values. This is converted into a JSON string via the function JSON.stringify(), and the JSON text is sent to the server, where the received JSON is converted into a Java object. Once in Java format we can marshal it to native IEEE 1278.1 DIS for distribution on the server's network, or send it to other web clients in JSON format.

3D is in this case handled by GLGE, a Javascript framework that utilizes WebGL and provides

scene graph functionality, along with the ability to load Collada models. A screen capture of the application is shown in figure 3.

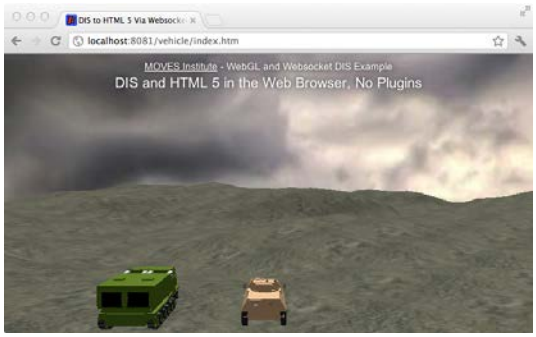

Figure 3. Vehicles Displayed in Web Browser

The web browser, Google Chrome in this case, is displaying a scene using the GLGE framework. The vehicles displayed on the screen can be moved by ESPDUs in JSON format delivered by the server over WebSockets.

Javascript and Web Application Architecture Issues. Older versions of Javascript were not noted for speed—it is, after all, an interpreted text-based scripting language—but market competitive pressures have changed this. Since so many web applications use Javascript and Ajax there has been intense interest by browser vendors in improved performance.

The performance we observed is quite good. A Windows 7 laptop with a Core 2 Duo T9600 CPU at 2.8 GHz with 4 GB of memory can decode over 500 JSON-ized ESPDUs per second while consuming less than 10% of CPU. Sending DIS PDUs from the server side is somewhat more expensive; converting Java objects into JSON takes more time, and converting approximately 1000 ESPDUs per second to JSON and then sending them over a WebSocket consumes most of one core of a four-core 2.8 GHz Intel Xeon. On the web browser side the gating function is graphics update speed, not network performance or the CPU cycles devoted to decoding JSON-format DIS. As with most NVE designs, careful design tradeoffs need to be made to balance the CPU budget between networking, graphics, physics, dead reckoning, artificial intelligence, and other tasks. But it appears that the CPU necessary for JSON-format DIS communication over WebSockets is well within the capabilities of modern desktop and laptop CPUs.

Working inside a web page creates some unique problems. One of these is that the simulation state rests on the foundation of the web page within the browser, which is a foundation of sand at best. The Javascript in the web page begins execution when the page loads. The Javascript code has state information, such as the position of the entities it controls, the position of other entities it has heard about from the web socket, and camera position. If the user hits the "reload page" button all this state goes away and must be rediscovered by the application or recreated by the user. The user has restarted the application by the act of reloading the page. There is no easy way to prevent users from doing this. In effect, a gigantic ON/OFF button is sitting in front of the user, and some may be tempted to use it.

The Javascript programming paradigm is *partially* single-threaded. All access to the web page's Document Object Model (DOM), drawing on the screen, or user interface work must be done from a single thread, and this can make it difficult to implement some features such as physics, AI, or dead reckoning that should run concurrently with user interaction. This can be worked around to an extent via the use of a programming feature called Web Workers, a draft standard promulgated by the W3C [13]. Though the standard is still in draft form a number of browser vendors have already implemented it. Web Workers can run in separate threads, and can be offloaded to whatever cores are available on the CPU, but cannot concurrently access the DOM. Instead, Web Workers can post notifications to the main thread that they have data available that should be displayed. With multi-core CPUs this feature is a very valuable for scaling, and for making tradeoffs between features such as physics and networking.

Because the Javascript NVE rests on top of the Javascript layer of many different browser vendors, programmers must be alert to implementation incompatibilities. Mozilla, the organization that created Firefox, decided the name of their implementation of WebSockets should be "MozWebSocket". This resulted in the compatibility check shown in figure 4.

```
if(window.WebSocket)
 websocket =
   new (WebSocket(aUri, “nve”);
else if(window.MozWebSocket)
 websocket =
  new MozWebSocket(aUri, “nve”);
else
   console.log(“Websockets not
supported”);
```

Figure 4. Platform Incompatibilities

This example is not particularly difficult to detect and work around, but shows that the NVE depends on Javascript implementations that may not be entirely consistent between vendors.

For security reasons Javascript runs inside of a “sandbox” in the web browser and therefore has very limited access to the disk and to user information stored on the client machine. Migrating the information needed by the NVE to the server side can mitigate the limitations caused by the sandbox.

Despite these drawbacks, Javascript has some interesting possibilities. The Javascript code is initially kept on the server side and downloaded to the client, which makes modifying the NVE very easy in comparison to redeploying a desktop application. Instead of tracking down every installation, deleting it, and installing a new application, the administrator can simply deploy the code on web server and ask users to reload the page.

Javascript also meets the definition of “mobile code.” While we have not made use of the capability in this iteration of the software, it is possible to download code associated with entities in addition to simply moving the graphical representation of the entities in the environment in response to WebSocket messages. Elements downloaded from the server could include AI, agents, and other behaviors.

Application designers should be alert to the amount of data necessary for a web page load. The simple scene shown above had a total web page size of 4 MB. Most of the data consisted of models and textures. The GLGE Javascript 3D scene graph framework was only 300 KB, while the MLRS and Stryker models, which were saved in the Collada format, had a combined size of over 3 MB; the remainder of the page size consisted of various textures and images. The Javascript code makes up only a small percentage of the overall page content. There are well-known techniques for reducing the impact of large page loads. For example, browsers cache large files on the local disk to prevent having to repeatedly download them via the network. Other techniques such as Content Distribution Networks (CDNs) can be used to speed downloads at sites with poor network connectivity.

Future Work. We plan to implement the 3D graphics portion of a NVE with X3D and X3DOM.

Scalability on the server side is an important issue. A single servlet on the server side may not have adequate CPU to operate as a data hub once traffic gets high enough. This can be addressed by clustering servlets in the data center, so multiple nodes handle traffic.

Mobile devices are increasingly important. Recent mobile browsers generally implement WebSockets and WebGL, though less consistently than in the desktop environment. Mobile devices have less available bandwidth, power, and graphics processing ability than desktops or laptops. This can be addressed by downloading device-dependent scenes. For example, an iPhone might be provided lower-resolution and smaller models than a desktop, based on what the server detects as the client.

A NVE should allow extensibility, and Javascript meets the definition of mobile code. Designers should be able to load content into the scene that is capable of operations such as physics, dead reckoning, and artificial intelligence. However, this requires that well-defined interfaces be defined so that new modules can seamlessly plug in.

Endnotes

[1] Haik, Eyal, Barker, T, Sapsford, J. Investigation into Effective Navigation in Desktop Virtual Interfaces. Web3d ’02: Proceedings of the Seventh International Conference on 3D Web Technology, pp 59-66.

[2] Dickey, M. J. (2005). Three-Dimensional Virtual Worlds And Distance Learning: Two Case Studies Of Active Worlds As A Medium

For Distance Education. British Journal of Educational Technology 36 (3) 439–451.

[3] http://www.khronos.org/webgl/, retrieved 5/7/2012

[4] http://www.x3dom.org/, Downloaded 7/5/2012

[5] http://www.glge.org/, Downloaded 7/5/2012

[6]http://mrdoob.github.com/three.js/, Downloaded 7/5/2012

[7]http://www.ambiera.com/copperlicht Downloaded 7/5/2012

[8]http://tools.ietf.org/html/rfc6455 Downloaded 7/5/2012

[9] http://www.w3.org/TR/websockets/, Downloaded 7/5/2012

[10] Institute for Electrical and Electronics Engineers. IEEE Standard for Distributed Interactive Simulation: Application Protocols. 1998. ISBN 0-7381-0174-5

[11] http://www.ietf.org/rfc/rfc4627.txt?number=4627, Downloaded 7/5/2012

[12] McGregor, D, Brutzman, D, *Open-DIS: An Open Source Implementation of the DIS protocol for C++ and Java*. Proceedings of 2008 Fall Simulation Interoperability Workshop, 2008.

[13] http://www.w3.org/TR/workers/ Downloaded 7/5/2012