jifi-eai 2012
Jornadas de Investigación Encuentro Académico Industrial
UNIVERSIDAD CENTRAL DE VENEZUELA FACULTAD DE INGENIERÍA
26 - 30 noviembre 2012

Web visualization of 3D medical data with open source software

Esmitt Ramírez J.[1*], Ernesto Coto[1*]
[1]*Centro de Computación Gráfica, Escuela de Computación, Facultad de Ciencias, Universidad Central de Venezuela, Los Chaguaramos 1041-A. Caracas, Venezuela.*
[*] esmitt.ramirez@ciens.ucv.ve, ernesto.coto@ciens.ucv.ve

ABSTRACT

Every day more applications migrate from off-line systems to web applications available worldwide. Many of these applications require displaying 3D data and allowing user interaction with them. Websockets provides the basis to develop these applications over the web, to run on any platform. This paper proposes four software architectures for developing web applications for 3D data visualization based on current open source websockets technology. One of these architectures is implemented and described in detail. Test and results for 3D meshes and volumes are presented and discussed. The system can be used to display medical datasets and medical models on real-time over the web, potentially contributing to the Telemedicine area.

Keywords: Web Visualization, 3D, Medical Data, Websockets, Open Source.

RESUMEN

Cada día más aplicaciones migran de sistemas sin conexión a aplicaciones web disponibles en todo el mundo. Muchas de estas aplicaciones requieren el despliegue de datos 3D y la interacción del usuario con los mismos. Los websockets proveen la base para desarrollar estas aplicaciones para la web, para ejecutarse en cualquier plataforma. Este trabajo propone cuatro arquitecturas para el desarrollo de aplicaciones web para la visualización de datos 3D, basadas en la tecnología actual de código abierto de websockets. También se describe la implementación detallada de una de las arquitecturas. Finalmente, se presentan y discuten pruebas y resultados para mallados 3D y volúmenes. El sistema puede ser utilizado para desplegar volúmenes y mallados médicos en tiempo real sobre la web, contribuyendo potencialmente al área de Telemedicina.

Palabras Clave: Visualización Web, 3D, Datos Médicos, Websockets, Código Abierto.

INTRODUCTION

Nowadays, websockets are included in the HTML5 specification which offers a communication based on sockets. This communication is bidirectional and full duplex over the TCP protocol. Thus, a simple server can send data towards clients without creating a long-polling scheme or using any browser plugin. The websocket open source technology represents a real advantage over that of other servers based on push technologies for the HTTP protocol, e.g. the Comet technology [1].
Using websockets, when a handshaking process is complete between a server and a client, messages can be sent in both directions without exchanging extra data. This reduces the required bandwidth, improving the performance of the application. Therefore, a server can transfer new

data to clients without making additional push operations, consequently reducing the server's processing time.

Several web browsers and web servers currently support websockets. Particularly, as shown in [2], the websocket implementation for the open source Node.js server called Socket.IO, has become very popular, as a flexible and versatile option to the real-time communication between a browser and a server.

As stated by Griffin et al. [3], the platform created by a Node.js server is completely event-driven with a non-blocking input/output mechanism for creating scalable programs over a network, running on the server side under an asynchronous model. Node.js runs over the V8 Javascript engine developed by Google, written in C/C++ and Javascript [4]. The V8 engine is a virtual machine to applications requiring fast response times and supporting HTTP, TCP and DNS protocols. Node.js also runs over several different operating systems and its engine handles all incoming connections in a single execution thread. This last factor distinguishes it from other servers. Then, Node.js is efficient for applications which require a persistent client-browser connection for a constant and rapid transfer of data between them.

A clear example of these applications is 3D data web visualization, where the rendering process must be performed in real-time, while capturing all user interaction. According to Martin [5], methods for rendering 3D data can be classified according to where the rendering takes place, in: client side methods, server side methods and hybrid methods. Behr et al. [6] suggested a classification according to the technology used for rendering: with or without plugins. The plugin solution has been widely used for many years. This is because all applications manage their data depending on their final goal, such as offering different image qualities, different compression rates, and others.

Many different web visualization systems have been proposed over the years, involving various areas of research. For instance, Koller et al. [7] developed a remote rendering system of 3D data for private data protection against piracy or misuse. The work of Dos Santos and Pedrini [8] is another example, where four architectures are presented for on-demand remote 3D rendering of geometry using clients with low graphics processing capacities. On each architecture, the quality of the final images varies in accordance with the employed technique. A natural aspect of web visualization, dealt with in many research papers, is the rendering of large data. Meyer et al. [9], for instance, proposed the interactive rendering of large data in full resolution using hierarchical rendering techniques over a Java applet with VRML2 files. Similarly, Okamoto et al. [10] proposed an interactive rendering system for large 3D mesh models based on the cloud computing concept. Other works are focused on geographical data [11], training and education [12][13], urban models [14][15], medical data [16][17] and others.

With all these contributions in mind, in this paper we present a multiplatform system for real-time visualization of 3D data over the web, based on the open source Node.js server and HTML5, that integrates current web technology with standard rendering algorithms. The system could be used to display volumetric datasets and geometric models on real-time over the web.

In the next section we present, according to our point of view, four different possible architectures that could be used to render 3D data in a web browser on the client side. Later on, we present the implementation of one of these architectures. Then, we present the results of the implementation, conclusions and possible future works.

STUDY OF POSSIBLE ARCHITECTURES

With current technology, it is possible to show 3D data without requiring a specialized graphic

hardware offering high client-server interactivity. This can be accomplished using the server as a processing center for performing the off-line 3D rendering and a web browser to display the final result. In the following subsections, we present four different possible types of software architectures to render 3D data on a web browser. Figure 1 shows all proposed architectures.

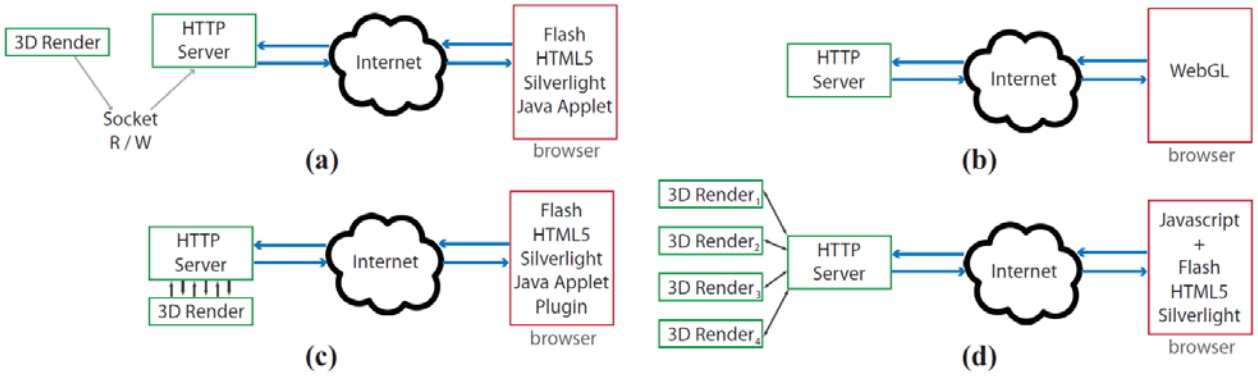

Figure 1. The four proposed architectures.

Architecture #1

Figure 1(a) shows a graphical scheme for this type of architecture. This architecture includes a 3D Render module which performs all operations to generate a 2D image out of the 3D data. Ideally, this should be implemented on the GPU for optimal performance, although a software implementation could be possible as well. This module must have access to a database of 3D data, usually located in the same physical location of the HTTP server, in the server side. After the 2D image is generated, its width, height and pixel values are sent to a networking port using a socket connection. Both the 3D Render module and the HTTP server should be constantly connected to the port.

The client side, a browser, is connected to the server through the Internet. Once the server reads the image from the port, it sends the image to the client which uses any RIA (Rich Internet Application) technology to display the image and allow interaction with the user. The client side could be based on Flash, Java, Silverlight or other technologies.

The disadvantage of this architecture is that the communication between the 3D Render module and the server could be complex. Frequently they use different programming languages and this could be a major problem. A possible solution is introducing an intermediate language that both parties can understand, for example Thrift (*http://thrift.apache.org/*).

Architecture #2

This architecture is more simple than the previous because the server only receives the client's HTTP requests and answers them. Then, the browser must render the 3D data by itself. Figure 1(b) shows a scheme of this architecture.

The 3D data is stored on the server side so the client can access them directly using some URI or URL address. Another widely used option is sending from the server a low resolution version of the data. This version can vary according to the compression methods, the size of the original model and the available bandwidth.

On the client side, there must be a technology capable to render 3D data and display it in the browser. A popular option is using WebGL [18], an OpenGL implementation for web clients. With WebGL it is possible to render the data without using plugins. However, it is not supported

by many browsers. In the near future, the majority of browsers will support WebGL over a HTML5 canvas. Notice that, WebGL performs rendering using the client's graphics card, so rendering times always depend on the quality of the client's hardware. Java3D is also a valid option but requires a plugin installation.

Architecture #3

Figure 1(c) presents the scheme of the architecture. Notice that the 3D Render module is integrated into the HTTP server. This way, the processing and the communication associated with the rendering is completely built into the server, this means they are both running in the same programming language or using shared libraries, and they are both totally dependent on each other.

Then, the programming language must be carefully selected considering the efficiency of the graphics pipeline. Languages such as Javascript, C++, Java, Ruby and so on, are good candidates for containing the 3D Render module and work as a HTTP server as well. Ideally, the server must be a high-performance computer capable of connecting to the Internet at high speeds.

As well as with Architecture #1, some kind of RIA technology is required on the client side. Similarly, the client can require a plugin to display the 3D data sent by the server. The main goal is to ensure the real-time data transference.

In this architecture, all the 3D rendering processing is performed on the server. On the client side the procedure is simpler from the programmer point of view, since just sending the image from the server to the client suffices to display the image in the browser.

Architecture #4

The main idea, see Figure 1(d), consists in doing the rendering using some processing cluster or a high-performance computer grid. Thus, the rendering process is parallelized distributing the computation load in different computing units (CU). The computing units should ensure a low-latency to obtain a better performance. Each computing unit should produce one part of the final image.

Note that the CUs must be managed and controlled by the HTTP server or a built-in process. The server must coordinate that all parts of the final image are correctly produced. These tasks can generate an overload on the server. All image parts are then sent to the browser, which must compose them into the final image.

Since this architecture proposes an approach where the parallelism of data is relevant, the data coherence and concurrency can be exploited.

IMPLEMENTATION

For our implementation of the 3D web visualization system, we chose Architecture #3 because we can use Node.js as web server and exploit its advantages. Additionally, with this implementation we fill a gap in the literature, since to our knowledge there are no similar works published.

Our approach consists in creating a HTTP server to load a 3D model or volume and rendering a 2D image out of it. After that, this image is sent to a browser where the user can interact with it. Every time the user performs an action (rotation, zoom, etc.) a set of commands and parameters is sent to the server, which applies the operation and returns a new 2D image. This clearly corresponds to the aforementioned Architecture #3.

As can be seen in Figure 2, all 3D data to be managed are stored in the server. On the server side,

there is also an extension developed using Node.js for performing the rendering. In the client side, the browser receives the image and displays it using a HTML5 canvas component. The communication between server and client is done using Socket.IO, the websocket implementation of Node.js. The communication protocol is based on TCP rather than in HTTP.

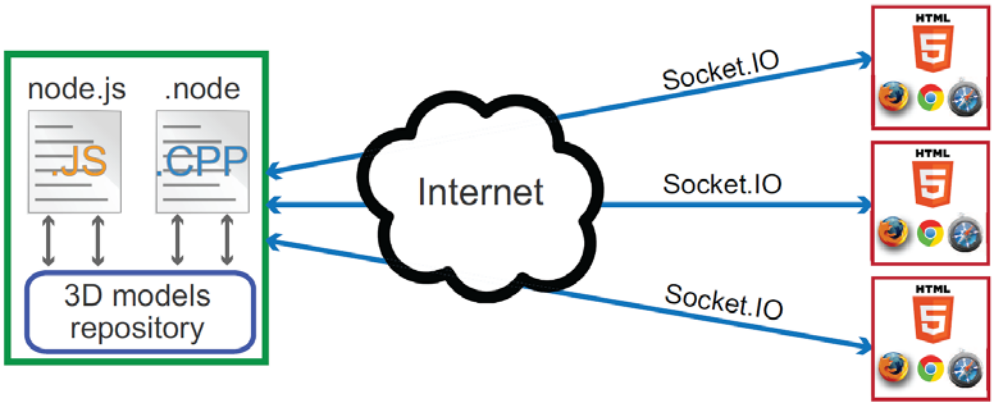

Figure 2. Proposed implementation following Architecture #3.

Note that by using Node.js the server side code is written totally in Javascript. This same source code can be used on servers with different operating systems, meaning that our approach is multiplatform. For simplicity and portability the execution of the rendering is performed in the CPU. Although we know that we could obtain faster response times using the GPU, this is left as future work.

Now, with this implementation, it is possible to separate the server side functionalities in two modules, which communicate with each other: **1)** Communication with the client and **2)** Rendering of 3D data. Also, in the client-side three different modules can be distinguished: **1)** Communication with the server, **2)** Image rendering and **3)** User interaction. In the next subsections, we explain the details around both the server and client side and each of its modules.

Server side

As mentioned before, our server code is written in Javascript, which is the language we use to handle the communication. However, we use an extension module of Node.js developed in C/C++ for the 3D rendering. This is possible since November 2011, where Node.js establishes native support libraries for the development of extensions on Windows using Visual C++.

The first module deals with the communication with the client. The server creates a Socket.IO instance for the HTTP software that runs on it. This instance manages a handler that waits for any client connection. This handler receives as parameter a string in JavaScript Object Notation (JSON) format which indicates the operation to execute. When a client requests an image resulting from the 3D rendering process, the server stores the image and sends it using Socket.IO. The rendered image is stored in PNG format in the server, but before sending it to the client, it is first codified in base 64 so that this process does not have to be made by the client. The result of the codification is a string, which is then sent using Node.js. When the image is received by the client, it can be displayed directly in the HTML5 canvas without any previous processing.

The second module deals with the rendering of the 3D data stored on the server side. A standard graphics pipeline is used for the rendering. The source code for all processes is written in C/C++. From that code we generate a Node.js extension in order to be used with the rest of the system.

The viewport dimensions for the rendering corresponds to the final image dimensions in RGB color space. The implementation supports both geometric and volumetric 3D data, meaning it can render meshes and volumes. The volumetric data is given in 8-bit raw format. The meshes are stored in OFF format. The volumetric data is rendered using the ray-casting technique presented by Krüger and Westermann [19], but completely implemented in the CPU. After the rendering is finished, the resulting image is sent to the client.

Client side

The client side consists of a browser with support for a HTML5 canvas component for displaying the 3D rendered image. The client-server communication is through websockets with Socket.IO, and the user interaction is carried out capturing mouse events with Javascript.

As aforementioned, the client side is composed by three modules. The first is related to the communication with the server. On the client side, a Socket.IO instance is created to wait asynchronously for the reception of messages from the server. This message is codified in base 64. Similarly, when the user interacts with the canvas, the Socket.IO instance sends a message to the server indicating the type of interaction and its parameters. For each message sent a response is expected, this is why the communication under Node.js is based on TCP.

The second module, the image renderer, consists on the process of displaying the received image into the canvas. The data stream received from the server represents the image and it is used inside a function called *OnDraw*. This function must be invoked repeatedly on each time interval to update the displayed image. To do this, Javascript provides a function called *setInterval* which receives the function to be invoked and the time to do so.

The third and last module, manages the user interaction using the mouse. When using the canvas component one can define functions to caught mouse and keyboard events in an asynchronous way. Then, three basic functions were created to capture mouse button-presses, button-releases and movements. Each function executes instructions on the client side to manage the canvas.

TESTS AND RESULTS

The system was tested over two different operating systems for the server and the client, Windows and Linux. The extension developed for the 3D rendering was compiled using the g++ and Visual C++ compilers. The resolution of the final image was set to *512 x 512* pixels. All tests were run in a computer with a Quad 2 Core 2.4 GHz with 4 GB of memory (for client and server). We measured rendering times and the compression rate of the rendered image in PNG format. Two meshes and two volumes were used, see Table 1 for details.

Table 1. 3D data used to test our approach.

Number	Name	Characteristics
1	gauss	Mesh – # vertex: 2500; # triangles: 4802
2	horn	Mesh – # vertex: 3200; # triangles: 6162
3	head	Volume – 256 x 256 x 94 voxels; 8-bit
4	bonsai	Volume – 256 x 256 x 256 voxels; 8-bit

First, we measured the rendering time of our CPU approach in two parts: the process of obtaining the resulting image and the process to codifying the result image in a string. The results are shown in Table 2, where column *Render time* shows the average time of the graphics pipeline and

column *Image time* shows the average time to convert the PNG image to a base 64 string.

Table 2. Rendering times for the 3D data, in seconds.

Name	Render time	Image time	Total time
gauss	0.0087 s	~ 0 s	0.0087 s
horn	0.0099 s	~ 0 s	0.0099 s
head	0.1373 s	0.0391 s	0.1764 s
bonsai	0.2069 s	0.0357 s	0.2426 s

With the meshes, the *Image time* is infinitesimal, i.e. less than 10^{-5} seconds, which is considerable less than for the volumes. The *Render time* is also higher with the volumes, because the process used in [19] requires more computation than rendering a simple mesh. With these times, displaying the volumes can reach up to 5 fps in the case of the *head*, and 4 fps for the *bonsai*.

The second test performed was changing the compression of the PNG file. For this, the Zlib (*http://zlib.org*) library was used with the *DEFLATE* stream compression. The options used to generate the final image ranged from compression level 1 (faster, low-quality) to level 9 (slower, high-quality) and no compression at all. After comparing the visual results, and taking into account that image generation does not produce a time overhead in the algorithm, we observed that a compression level of 6 is enough to achieve good visual results. With this value, the result is visually similar to not compressing the image. The difference in computation time between level 1 and 9 is around 10^{-3} seconds.

Rendering times vary according to the number of pixels displayed in the viewport (depending on rotations and translations). For example, rendering the *head* model can take up to 0.3264 seconds when it occupies 80% of the viewport, and 0.1563 s when it occupies only 50% of the viewport.

CONCLUSIONS AND FUTURE WORK

This paper presented four different architectures, all based on current technology, which could be instantiated for many implementations to render 3D data on a web browser. We also implemented a system based on Architecture #3, which is multiplatform, uses open source software and standard rendering techniques.

The system can render both volumes and 3D meshes, so a possible application could be displaying medical data over the web, such as datasets from a PACS server (first allowing the system to read DICOM files) as well as rendering meshes corresponding to implants or other 3D medical objects. This could be a potential contribution to the area of Telemedicine.

Our approach achieves interactive rendering times of 4-5 fps in average for the volumes and even better for the meshes. As expected, rendering volumes takes more time than rendering meshes, although this might depend on the complexity and size of the data. Further tests are needed to explore this possibility.

We have also shown that with our approach one can compromise the quality of the final image while still obtaining good results without affecting the performance. Additionally, note that the times to codify the image are negligible in all cases. In the future, system performance can be improved by rendering the data using the GPU or implementing our CPU approach using multi-threading or multi-processor technology. Also, it would be interesting to test the four presented architectures using the same input data and then compare the results.

REFERENCES

[1] Crane, D., McCarthy, P. (2008) Comet and Reverse Ajax: The Next Generation Ajax 2.0. Apress.
[2] Thompson, M. (2011) Getting Started with GEO, CouchDB, and Node.js. O'Reilly Media.
[3] Griffin, L., Ryan, K., De Leastar, E., Botvich, D. (2011) In: Proc. of IEEE Symposium on Computers and Communications (ISCC) 2011, "Scaling Instant Messaging communication services: A comparison of blocking and non-blocking techniques", pp. 550-557.
[4] Lerner, R. M. (2011) At the forge: Node.JS, Linux Journal, article 6. Belltown Media, USA.
[5] Martin, I. M. (2000) Adaptive Rendering of 3D Models over Networks Using Multiple Modalities, Tech. Rep. RC21722, Watson Research Center.
[6] Behr, J., Eschler, P., Jung, Y., Zöllner, M. (2009) In: Proc. of the 14th International Conference on 3D Web Technology, "X3DOM: a DOM-based HTML5/X3D integration model", pp. 127—135. Germany.
[7] Koller, D., Turitzin, M., Levoy, M., Tarini, M., Croccia, G., Cignoni, P., Scopigno, R. (2004) Protected interactive 3D graphics via remote rendering, ACM Transactions on Graphics (TOG), vol. 23(3), pp. 695-703. ACM, USA.
[8] Dos Santos, M., Pedrini, H. (2009) Master's Thesis: Arquiteturas para Renderização de Cenas Tridimensionais em Computadores com Baixa Capacidade de Processamento Gráfico. Universidade Federal do Paraná, Brazil.
[9] Meyer J., Borg, R., Hamann, B., Joy, K., Olson, A. J. (2003) Network-Based Rendering Techniques for Large-Scale Volume Sets, Hierarchical and Geometrical Methods in Scientific Visualization, pp. 283-295. Springer-Verlag.
[10] Okamoto, Y., Oishi, T., Ikeuchi, K. (2011) Image-Based Network Rendering of Large Meshes for Cloud Computing, International Journal of Computer Vision, vol. 94(1), pp. 12-22. Kluwer Academic Publishers, USA.
[11] Huang, B., Lin, H. (1999) GeoVR: a web-based tool for virtual reality presentation from 2D GIS data, Computers & Geosciences, vol. 25(10), pp. 1167-1175.
[12] Petersson, H., Sinkvist, D., Wang, C., Smedby, O. (2009) Web-based interactive 3D visualization as a tool for improved anatomy learning, Anatomical sciences education, vol. 2(2), pp. 61-68.
[13] Mendes, C., Luciano, S., Bellon, O. (2012) IMAGO Visualization System: An Interactive Web-Based 3D Visualization System for Cultural Heritage Applications, Journal of Multimedia, vol. 7(2), pp. 205-210. Academic Press.
[14] Royan, J., Gioia, P., Cavagna, R., Bouville, C. (2007) Network-Based Visualization of 3D Landscapes and City Models, IEEE Computer Graphics and Applications, vol. 27(6), pp. 70-79. IEEE Computer Society Press.
[15] Zhang, L., Han, C., Zhang, L., Zhang, X., Li, J. (2012) Web-based visualization of large 3D urban building models, International Journal of Digital Earth, pp. 1-15. Taylor & Francis.
[16] Engel, K., Westermann, R., Ertl, T. (1999) In: Proc. of the Conference on Visualization'99, "Isosurface extraction techniques for Web-based volume visualization", pp. 139-146.
[17] Chern-Yeow, D., Choe, Y. (2008) In: Proc. of the IEEE/EG International Symposium on Volume Graphics, "Stereo Pseudo 3D Rendering for Web-based Display of Scientific Volumetric Data", pp. 73-80.
[18] WebGL (Septiembre 2012) http://www.khronos.org/webgl/. Khronos Group.
[19] Krüger, J., Westermann, R. (2003) In: Proc. of the 14th IEEE Visualization 2003, "Acceleration Techniques for GPU-based Volume Rendering", pp. 287-292.