Web Based 3D Visualization for COMSOL Multiphysics®

M. Jüttner*[1], S. Grabmaier[1], W. M. Rucker[1]
[1]University of Stuttgart – Institute for Theory of Electrical Engineering
*Corresponding author: Pfaffenwaldring 47, 70569 Stuttgart, Germany, ite@ite.uni-stuttgart.de

Abstract: A web based visualization solution for three dimensional simulation results of COMSOL Multiphysics is described. With it, mobile clients with low bandwidth as well as desktop computers with high bandwidth connections get able to show simulation results without the need of installing specialized software. To do so, a modern web browser is required. Both touch gestures and common input techniques supports real time interaction with the visualization. The required web application, the web server and the data evaluation at a COMSOL Server is described. Details about data handling and rendering are also discussed. The performance of the system is shown for different computer systems and different simulation results.

Keywords: HTML5, Mobile Clients, Visualization, WebGL, web sockets.

1. Introduction

An important point for interpreting results of numerical simulations is the visualization. Same holds for transferring knowledge about hardly conceivable relations. So especially in the fields of teaching, product or research presentations and in future multiphysics problem solving environments [1] an easily available and lightweight visualization solution is expected. Mobile devices and their connection to the internet represents a chance to provide a solution for this needs. While modern internet clients run on different operation systems, multi-platform development is necessary. For small internet applications, a combination of the Hypertext Markup Language (HTML), the Cascading Style Sheets CSS and JavaScript represents an open standard for developing and porting the developed applications to different platforms. The performance of these internet applications increases with new introduced web technologies like HTML5 [2]. Nowadays even native HTML applications are possible [3]. Although the final specification of HTML5 and related web technologies are not released yet, common web browser support them. So web pages even changes from former static documents to web based applications. As prerequisite a powerful 3D Web based Graphics Library (WebGL) was needed. WebGL offers a JavaScript APplication Interface (API) for graphical programming. It allows web application to take advantages of the local graphic processing unit (GPU) [4].

In the following section, the outlined internet technologies are used to create a system, able to display COMSOL's 3D visualization plots. The system setup as well as the required components are described. Section 3 shows the performance of the system on examples out of the COMSOL Model Library and on both, mobile and desktop clients. A conclusion is given in section 4.

2. Visualization system

To show results of a simulation within a client's web browser the components shown in fig. 1 are needed.

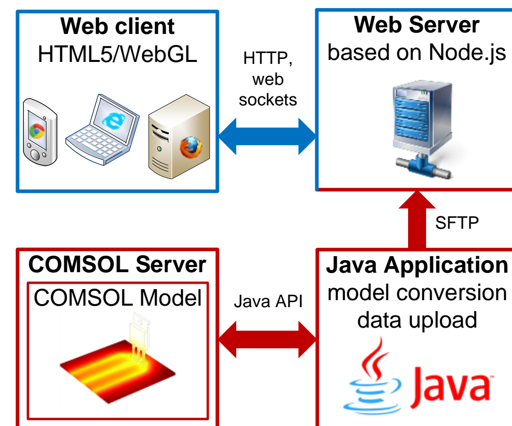

Figure 1. System architecture

The web client represents the user interface. Here the visualization of the results are displayed and user input is generated. Because of different operating systems and individually chosen web browsers at the web client, a universal interface is needed. This interface is build using web sockets. It provides a bidirectional communication with a web server to allow a dynamic reload of new information. The web server provides the web application for visualization. Here Node.js is used as web server providing the web application and all data necessary for visualization. It is based on Google's JavaScript Runtime Environment. A package manager and free available development tools make it easy to handle. To get the required

data out of a COMSOL Server and into the web server, a java application is created. Using the COMSOL Java API (CJAPI) this software extracts necessary data and commit it via a Secure File Transfer Protocol (SFTP) to the web server. Currently the connection between the web server and the java application is unidirectional. Section 4 outlines additional features permitted by a bidirectional connection. The following chapters describe relevant details for implementing this system.

2.1. Data extraction and conversion

The CJAPI provides direct access to the model object, containing all algorithms and data structures for a COMSOL model. The structure of the COMSOL model and especially the plot data is used in wide parts of the java application and the web application. The structure consists of two data elements. One binary element containing the raw data and one text based file containing all meta information about the plot data. The binary files structure is shown in fig. 2.

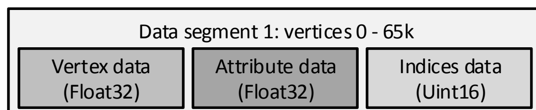

Figure 2. Structure of the binary plot data

Each data segment contains three different kinds of value. The vertex data containing all coordinates for visualization. Optional attributes for the corresponding vertices are stored in the next segment. The mapping between the vertex data and primitive elements (lines, triangles) are stored at the indices data segment. Since WebGL is used for visualization and it only supports the Uint16 data type for indices, data is split into segments of 65k vertices. WebGL also requires a draw call for each data segment. Using JavaScript as a well supported programming language at the Node.js web server [5], the eXtensible Markup Language (XML) based meta information file was converted to a JavaScript Object Notation (JSON) file. This allows a native handling of the models meta data within the web server.

2.2. Web and data server

To use only standardized web technology at the web server, SFTP is used for data exchange with the java application. Between the web client and the web server, the Hyper Text Transfer Protocol (HTTP) and web sockets are used. Web sockets allow upgrading the HTTP to a bidirectional connection. In contrast to the default Base64 encoded HTTP, web sockets directly transmits binary data. The transmission of type array buffer objects or binary large objects is possible. Using web sockets fundamentally increases the performance of the data transfer [6]. This is of special interest, because we deliver visualization data. There is no need to exchange the numerical results. The final web page is built at the client's web browser. This results in less logic within the web server but high input/output performance and scalability. An impression about the average data size of numerical plots for examples from the COMSOL Model Library is given in tab. 1.

Table 1. Data size

Model	*Plot type*	*vertices*	*Size*
Inductive Heating	Surface	10730	290 kB
Power Transistor	Surface	37108	898 kB
Power Transistor	Arrow volume	5079	120 kB

The process of establishing a connection between the web client and the web server is shown in fig. 3.

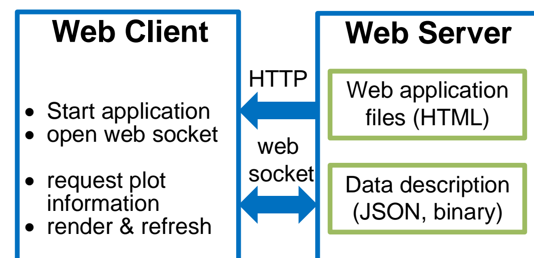

Figure 3. Data transfer between web server and client

First, the web server transfers the web application via HTTP to the web client. Then, the application opens and a web socket connection between the client and the server is established. From now on, all communication is done via this web socket. Two types of messages are implemented. The meta information exchange via JSON objects and the binary data as array buffer objects. Here array buffer objects represent the top class of typed arrays. Typed arrays allow different data types for numbers and native array

operations. Benchmarks have shown that the adequate usage of typed arrays leads to a better performance [7]. Typed arrays are also requested by the WebGL API to create buffer object for uploading data to the GPU.

2.3. Web Application

The purpose of the web application is the visualization of 3D data from COMSOL Multiphysics. To show a good representation of the results a fine granular mesh is needed. Especially for streamlines, isolines or isosurfaces plots have to look smooth. Therefore, a high density of vertices is require. This leads to a huge and strongly model and plot dependent amount of data that needs to be processed even for simple 3D visualizations. For example, the visualization of the surface temperature of the "Power Transistor" (COMSOL Model Library) consists of more than 30,000 vertices. Some of the vertices have geometrical meaning. Most represents coordinates for the visualization of the temperature values. To avoid a reimplementation of simulation result post processing, the renderer attached to COMSOL is replaced by a data interface to the web server. From the web server, the visualization data is forwarded to a web client and finally rendered in the web browser. The advantage by reusing COMSOL's data structure is the compatibility of all different plot types and the possible combination out of them. The extraction of the relevant plot data is described in section 2.1.

Rendering the data is done using WebGL, a strict subset of the Open Graphics Library (OpenGL) 2.0. Additionally, there is also mapping to Microsoft DirectX 9 and 11. In that way lots of devices are supported. WebGL uses the same render pipeline as OpenGL 2.0. Figure 4 shows the WebGL graphic pipeline with its programmable vertex and fragment shader.

Figure 4. Simplified rendering pipeline

Blue marked levels are programmed within the OpenGL Shading Language (GLSL). So data intensive structures can be handled by suitable and individually optimized developed shader programs. Using the pipeline all plots within COMSOL get available. To achieve good performance and make the application running on most devices, non-essential 3D effects for visualization are neglected.

The generic structure of the COMSOL visualization plots allows considering only a few different options while developing the web application render program. Tab. 2 shows the data structure for three different kinds of plot types.

Table 2. Different plot types
(nEle: number of elements, nVert: number of vertices)

Plot type	*Element dimension*	*Attribute type*	*Attribute dimension*
Surface	3 x nEle	colour	1 x nVert
Isolines	2 x nEle	[radius]	[1 x nVert]
Arrow Volume	null	vector	3 x nVert

For example, surface plots consist of triangles with one scalar value for each vertex. Mapping the values to a colour table (temperature, traffic light, etc.) a corresponding 1D texture is created. It contains 4 to 16 entries. The vertex shader scales the numerical value to values between 0.0 and 1.0. This value is used in the fragment shader to map the corresponding colour. The classical pong shader is implemented to achieve diffuse and spectral light effects. For lightning effects the normal vectors in each vertex is calculated. The render engine cyclically checks (every 16ms) if a user interaction has happened and relaunches a render process. Provided user interactions are translating, rotating and zooming. They are supported by mouse and touch gestures.

Plots like arrow volumes require additional calculation before visualization. Rendering arrow volumes, a vertex based arrow has to be generated for each value. Visualizing streamlines, the required three dimensional tubes are build using vertices and a width for the tubes. To avoid a separate draw call for each element, the elements are grouped. The required calculations are done using web workers, a sort of multithreading. They are used for calculating intensive data operations. Web workers are implemented on thread level and offer communication by message passing. In that way, blocking of the main thread and the user interface is avoided. The calculation operations can either been processed at the web server or at the web client. In case of processing them at the web server, lots of performance is available and the operations are performed only once. As

drawback, interaction or manual scaling is not possible. Additionally a higher data volume is transmitted. Here, the calculation of the elements is done at the web client using web worker and typed arrays. The calculation is done without troubling slowdown. Figure 5 shows the visualisation of an arrow surface and a surface plot.

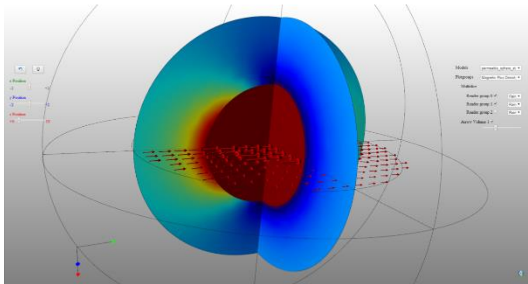

Figure 5. Screenshot of the web application.

3. Benchmarks

A first benchmark compares different programming languages by calculating the size, the orientation and the position for $1.5 \cdot 10^6$ arrows. Each arrow is built of 120 vertices based on vector data exported from COMSOL. Table 3 show the results for test *system a* with an Intel Core 2 Quad Central Processing Unit (CPU), a NVidia Quadro 600 GPU, Microsoft Windows 8, a screen resolution of 1680×1050 pixel, Visual Studio V.12 as C++ compiler with the -O2 flag, Java SE 1.7, HotSpot 24.51, Chrome: V 32.0 - 32 Bit and Firefox: V 27.0 - 32 Bit. Automatic clean up options like the garbage collector in java has been supressed.

Table 3. Benchmark JavaScript (JS)
(° using default arrays, * using typed arrays)

Environment	*Duration*	*Memory*
JS Firefox°	>5000 ms	1125.0 MB
JS Firefox*	∅ 2342 ms	402.4 MB
JS Chrome*	∅ 2454 ms	414.4 MB
Java 64 Bit*	∅ 1561 ms	421.0 MB
C++ 64 Bit*	∅ 1307 ms	396.5 MB

While programming languages like Java or C++ are still faster, the usage of typed array boost up the performance of JavaScript. The just in time compiler for JavaScript benefits from the high number of iteration over same code. The default JavaScript arrays are not suitable for such big data size. The reasons are no static memory allocation and Float64 as the only type for numbers. Using typed arrays reduces the memory consumptions to a size comparable to Java and C++.

Another benchmark shows the three dimensional visualization performance on different devices. Here *system a* is used again. *System b* uses is also a desktop system but without an additional graphics card. It uses an Intel Core 2 Duo CPU, an Intel X3100 GPU, a screen resolution of 1400×1050 pixel, Microsoft Windows 7 with service pack 1 and Chrome V 32.0 - 32 Bit and Firefox V 27.0 – 32 Bit as web browsers. *System c* runs on Android 4.2.2 with a screen resolution of 1280×696 pixel, a Mediatek MT8125 CPU and a PowerVR SGX544 GPU. Here the achieved frame rate is measured in Frames per Second (FPS). As reference model, the COMSOL example “Power Transistor” was used and some predefined interaction were processed too. Table 4 shows the results measured by the JavaScript performance monitor stat.js.

Table 4: Benchmark 3D performance

System	*FPS*
System a: Disabled GPU	∅ 11.7
System a: Enabled GPU	∅ 58.5
System b:	∅ 15.4
System c:	∅ 44.5

The first two results show that hardware acceleration by GPU is indispensable for 3D visualization. The CPU and the GPU of *system c* is used in smartphones and tablet computers. It also show a good performance. So running the introduced web application on mobile devices is possible. In the daily use, the frame rate is not as noticeable as during the measurements. The application is built to visualize simulation results. In that way slow frame rates only leads to a slow rotating or zooming process.

4. Conclusion

The visualization of COMSOL Multiphysics 3D simulation results by using standardized web technology has been shown. Additionally details for implementation were given. The visualization gets possible using a java application extracting rendering data from COMSOL Multiphysics and stores them on a web server. The web server hosts a web application. This web application establishes a bidirectional connection between the web server and a web client and allows a faster data transmission compared to HTTP. The connection is needed to transfer the rendering data to a web client. The necessary bandwidth is small, so actual mobile devices can handle it. At the web application, WebGL is used. It achieves a good visualization performance. This has been shown during a benchmark. Interaction with the visualization is possible by touch gestures and ordinary input techniques. To allow a user to recalculate or modify the simulation it is possible to establish a bidirectional connection between the web server and the COMSOL server by using the COMSOL API. In that case, security issues must be considered to provide a reliable system.

5. References

1. M. Juettner, A. Buchau, A. Faul, W. M. Rucker and P. Goehner, “Segregated Parallel and Distributed Solution of Multiphysics Problems using Software Agents,” *Conference on Electromagnetic Field Computation,* 2014.
2. R. C. Hoetzlein, “Graphics performance in rich internet applications,” *Computer Graphics and Applications,* pp. 98-104, 2012.
3. J. M. Wargo, Apache Cordova 3 Programming, Addison-Wesley, 2013.
4. P. Cozzi and C. Riccio, OpenGL Insights, CRC Press, 2012.
5. S. Tilkov and S. Vinoski, “Using JavaScript to build high-performance network programs,” *IEEE Internet Computing,* vol. 14, no. 6, pp. 80-83, 2010.
6. P. Lubbers and F. Greco, “HTML5 web sockets: A quantum leap in scalability for the web,” *SOA World Magazine ,* 2010.
7. S. Herhut, R. L. Hudson, T. Shpeisman and J. Sreeram, “Parallel programming for the web,” *Conference on Hot Topics in Parallelism,* 2012.