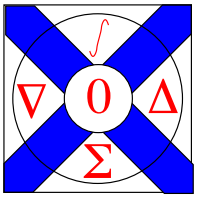

Computational Science Technical Note **CSTN-174**

Improving Platform Independent Graphical Performance by Compressing Information Transfer using JSON

T. H. McMullen and K. A. Hawick

2013

Interactive animation and other graphical applications are emerging as viable web services in a number of contexts including gaming and simulation. An important part of the performance tradeoff space is balancing the amount of processing work done on (usually slower) rendering clients against that on (usually faster) servers and therefore also the amount of fully or partially processed data that must be transferred across the network connection. This is particularly important when tablet computers and other lower end performance clients are used. Portable or platform independent graphics rendering can be achieved using web clients and we present software architectural ideas and prototypes using JSON and WebGL-based technologies and appropriate data compression and partial processing approaches. We give some performance data and a discussion of the implications for future high-performance platform independent graphics.

Keywords: JavaScript, JSON; graphics performance; platform-independence; compressed information; graphical web-services

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-174,
        author = {T. H. McMullen and K. A. Hawick},
        title = {Improving Platform Independent Graphical Performance by Compressing
                Information Transfer using JSON},
        booktitle = {Proc. 12th Int. Conf. on Semantic Web and Web Services (SWW'13)},
        year = {2013},
        number = {CSTN-174},
        pages = {SWW4052},
        address = {Las Vegas, USA},
        month = {22-25 July},
        publisher = {WorldComp},
        institution = {Computer Science, Massey University, Auckland, New Zealand},
        keywords = {JavaScript, JSON; graphics performance; platform-independence; compressed
                information; graphical web-services},
        owner = {kahawick},
        timestamp = {2013.04.22}
}
```


Prof Ken Hawick, Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand.

Technical Report CSTN-174

Improving Platform Independent Graphical Performance by Compressing Information Transfer using JSON

T.H. McMullen and K.A. Hawick
Computer Science, Massey University, North Shore 102-904, Auckland, New Zealand
email: timmy361@gmail.com, k.a.hawick@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

April 2013

ABSTRACT

Interactive animation and other graphical applications are emerging as viable web services in a number of contexts including gaming and simulation. An important part of the performance tradeoff space is balancing the amount of processing work done on (usually slower) rendering clients against that on (usually faster) servers and therefore also the amount of fully or partially processed data that must be transferred across the network connection. This is particularly important when tablet computers and other lower end performance clients are used. Portable or platform independent graphics rendering can be achieved using web clients and we present software architectural ideas and prototypes using JSON and WebGL-based technologies and appropriate data compression and partial processing approaches. We give some performance data and a discussion of the implications for future high-performance platform independent graphics.

KEY WORDS
JavaScript, JSON; graphics performance; platform-independence; compressed information; graphical web-services.

1 Introduction

The problem of implmenting complex graphical rendering applications [5, 9] on new emerging and especially mobile platforms [11, 13] is a challenging one. Whenever a new device is created, with it a new platform for applications to be deployed on tends to emerge. Graphical applications are no exception to this. This paper discusses the current limitations of platform independent graphical application, along with ways to overcome many of them. To achieve this platform independence, and help to improve performance, JSON files are used, along with binary files. As we wish for the applications designed to be platform independent we have based our research on web based technologies, and with it WebGL [1, 2, 4, 6] and JavaScript.

JavaScript is an interpreted programing language, and is heavily used to allow for browser to run client side scripts. JavaScript Object Notation (JSON) is a file format which is able to store data structures. These files can easily parsed and have their associated members, and variables accessed by another program. WebGL is the newest method to achieve platform independent graphics. It is based on OpenGL 2.0 [3, 15] and is implemented using a JavaScript [7, 8] API.

As WebGL uses OpenGL [10, 14] Shading Language (GLSL) to process objects, and create renderings, this allows for many existing shader models to be easily converted to work within this web based environment. One consideration to make when designing an application using WebGL is support, and limitation of hardware on mobile devices. Many smart phones, such as the Samsung Galaxy S3, or Google Nexes 7 are able to run WebGL applications, but due to fewer cores on the GPU, along with a larger limitation on bandwidth this leads to some negative effects. As such applications developed need to take this into consideration, and adapt to suit the run time environment.

This can be achieved by various means including reducing the precision of the variables within the shader, or by using fewer uniform variables on the shader.

The JSON file type is based on JavaScripts method for representing data structures, and is human readable. This relationship allows for the information contained within a JSON file to easily be parsed into a JavaScript variable. As these files integrate easily into JavaScript they become an ideal method for passing model files to WebGL for rendering. Converting from standard model files such as wavefont OBJ, and collada to a JSON as well as a converter to binary was necessary, and this has been achieved by using designing a plug in for some 3d modeling applications, along with other programs.

By using JavaScript embedded into HTML5, this allows an application to take advantage of many new features. These new features which can be utilized within a graphical application of this kind, includes the use of Web storage, in which it allows for some data which has previously being downloaded to be accessed without the need to redownload. Another use is that of offline web applications, these allow for an application to be run using previously accessed data, which may be stored in an SQL based database [12] , and cached data.

With all web based applications there is always the challenge of testing performance of a application vs the bandwidth required to transfer data to and from it. This can lead to the challenge of finding the ideal trade off between data transfer, and performance to create an application which is able to be loaded quickly, and runs as fast as it can on its deployed platform. Converting conventional model files into more web technology friendly file types helps to greatly improve both performance and bandwidth.

In this present article we investigate the architectural approach of creating an application using this web based method of implementing platform independent graphics. We create an application which will not become outdated as rapidly and will remain support as long as the devices are compatible with the HTML5 standard.

Our article is structured as follows: In Section 2 we talk about ways in which to setup and utilize WebGL along with different methods in doing so based on various file types. This includes passing data, along with how it is represented and stored within WebGL and JavaScript. Within Section 3 the ways in which optimization have been applied, along with the resulting improvements, in both the start up, and rendering times of the applications. Section 4 discusses the benchmarks, and how they were created, as well as some limitations which were incurred. The final Section 5 is the resulting conclusions of the implementations, and improvements found, along with some future work.

2 Implementation Strategies

This paper works to find ways of optimizing platform independent graphics, using WebGL by using customized file types. We first look into the ways of setting up the scene, and the models to be rendered. secondly the issue of loading in the required data, based on different file types, and finally the rate in which we are able to render objects to screen. A basic overview of these steps follows.

To render an object within WebGL, we need to know two things, firstly the positioning of each point which make up the object, and secondly how each of these points connect. Additional information includes normals and texture concordats, along with anything else needed to help improve the image quality of the produced scene. WebGL uses vertex buffer objects (vbo) to store the information for objects. To setup each of these VBOS the data for each file is read in, and bound to a respective buffer. It is these buffers which gone through and used to draw the object.

Loading in model files for using within WebGL can be done using XMLHttpRequest (XHR) these will load files and processes the data as required. Once a file has been loaded the data from within is parsed to the required vbo. Once the data is passed and the vbo setup correctly WebGL is able to draw an object to screen.

Many existing methods exist for loading models into a WebGL environment one of the more popular by using the JSON file formate, as it is implemented easily within JavaScript. The JSON formate allows for a model to be created and be processed into the required formate for it to be easily converted and passed into a vbo (vertex buffer object). The JSON method is popular as it greatly simplifies the process of loading an object, as it is setup correctly to be used within WebGL. By comparison to use a wavefont OBJ file requires a substantial set up time, as all the information needs to

be read, but then also converted into new arrays. This conversion process uses the face values in the OBJ file represented by a line in the file starting with "f", then using the values on which follow to determine how to make up that face. By using a JSON file this process is done offline and will not unnecessarily impede performance at start up. By comparing these two file types, it is clear to see that the JSON file will be faster to start up, but both are set up in the same way with each piece of information taking up its own buffer.

Once again we are able to improve upon this method by using VBOS and there ablility to have offsets assigned within them, this allows for a single vbo to contain multiple types of data, for example having one array containing all positioning data, texture coordinates and normals, while another stores the indices. By using these offsets within a vbo it removes the need to switch between buffers multiple times for each draw call. This can easily be achieved by using a binary file in place or alongside a JSON file. Algorithm 1 below shows the basis of the proccess in which is followed when wanting to reader a scene from this file type and method. Firstly the position attributes within the vertex shader are enabled, so it can be passed data, this is done for all attributes of the shader which will be passed data. The next step is to bind our position buffer, this is what is read in from the JSON file, and contains the positioning data for each point of the model. This information is then passed the the appropriate attribute on the shader to be processed when needed. This step needs to be repeated for each variable which you are using, in this case the texture data, along with the normals data. Finally the indices are bound and used to draw the elements above.

By switching to use a binary file to store the required information, the file size will decrease, improving on the start up time, and no longer needing to conver a string to floats also will improve this time. To use this method a model is exported into a binary file, where the first few bits tell you how each of the arrays stored are, along with what each part means, and where the offsets are, and how large they are. Once the file is loaded it is then split up and two new buffers are created from the array. Using JavaScript we are able to take this header information, and create new arrays based on the data from another and taking very specific parts. For this instance the requirements were one 32float array for storing the more complex information, and another 16 unsigned integer array for storing the indices. By using the offset within the buffers, the program will skip a set amount of bits within the array, and will know how and where to access the relevant information for rendering an object within WebGL.

Algorithm 1 Basic steps used in the process of drawing a frame. Requires a lot of addition work which could be simplified by using fewer arrays.

declare $position[]$, $texture[]$, $normals[]$, $indices[]$
enable position, within the shader program
enable texture, within the shader program
enable normals, within the shader program
bind position
link position array to the shader
bind texture
link texture array to the shader
bind normals
link normals array to the shader
bind indices
draw elements within the above arrays, based on indices

Algorithm 2 Similar to Algorithm 1, but by using fewer buffers, increase the speed at which each frame can be processed.

declare $faceBuffer[]$, $indices[]$
enable position, within the shader program
enable texture, within the shader program
enable normals, within the shader program
bind faceBuffer
link positions based on data from the faceBuffer
link texture based on data from the faceBuffer
link normals based on data from the faceBuffer
bind indices
draw elements within the above arrays, based on indices

Algorithm 2 above shows a similar process as in Algorithm 1, but with fewer steps. In this implementation, the attributes still are required to be enabled, but only the one buffer, in this case faceBuffer is used, because of this when the data is linked to the attribute on the shader it uses an offset to skip a set amount of information each time. By having this offset this allows for the one buffer to hold all the information on the object or model removing the need to switch between buffers and bind multiple buffers each call. The final step remains the same as it is in algorithm 1.

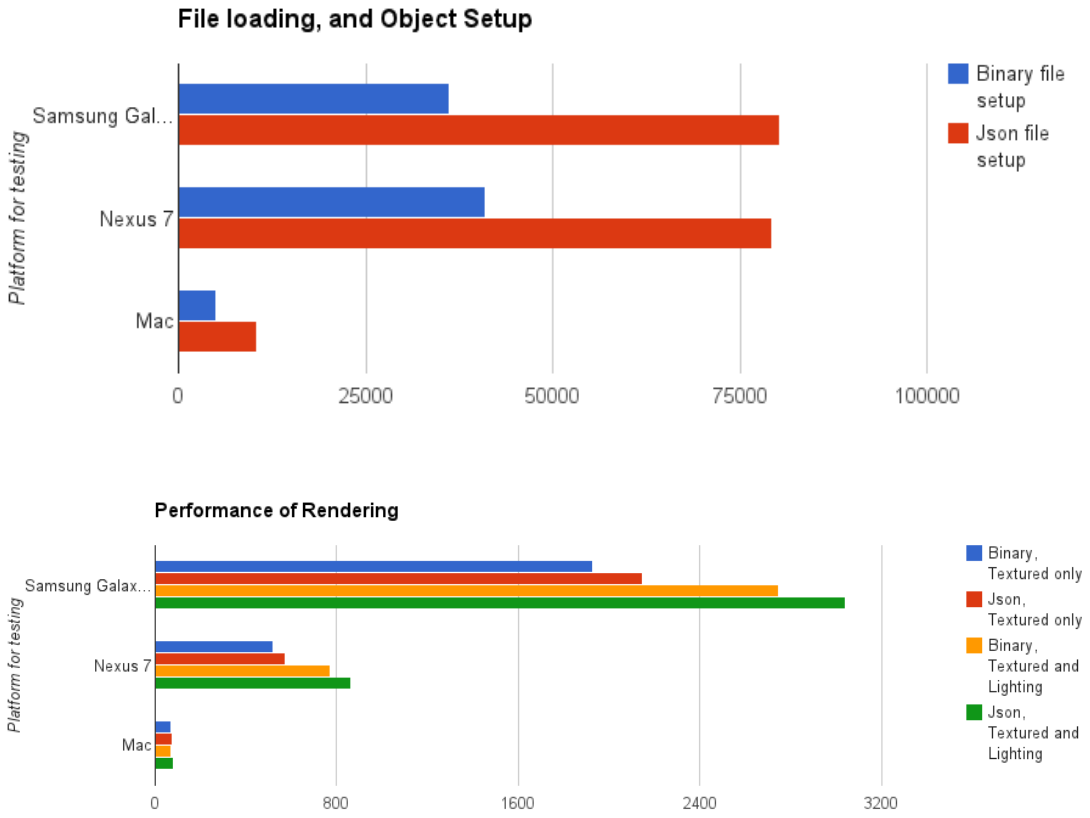

Figure 1: Time to load in and Setup 1,000 instances of the Utah Teapot (ms) - above; and time to render 10,000 frames – below

3 Selected Results

Figure 1 shows the times for various devices to load in and setup 1,000 instances of the standard Utah teapot model (above) and the times to render 10,000 frames containing it (below).

To test the performance increase by switching file types several aspects of the rendering application were timed. Firstly time taken to load in files and set up the arrays and as such the corresponding VBOS. The next tests consisted of testing the time taken to render an object in different conditions, such as with or with out textures or lighting, as each of these will alter the performance of the applications and GPU. The devices which were used for testing were a Mac Pro with two AMD 5770 GPUs, 8 Gbs Ram, and a quad core 3.2 Ghz processor, an Google Nexus 7, and a Samsung Galaxy S3. These devices were chosen as the performance difference between them and architecture can demonstrate how that will effect the testing environment, and as such an application which is implemented with this kind of technology

The first test took a wavefont OBJ file containing texture information along with normals, this was then converted into two different files, firstly a JSON file, with each array setup, then into a binary file. This resulted in turning a 94.8 kb OBJ file into a 36.9 kb binary file, significantly reducing the bandwidth required to transfer the file. This reduction in size, and improvement in format also helped in the time taken to setup a file. In testing the time taken to load 1000 instances of the same model, and set up the correct arrays, it took 19489ms to load and convert the OBJ files, by comparison the JSON files took 10415ms which is a good improvement. The binary file though was far superior to these two other approaches, in which

the time taken to load in 1000 files, and set up the correct arrays took 5092ms. This improvement in the start up time for binary files can be attributed to two aspects of it, firstly the reduction in file size, as this allows the file to be loaded much faster, and secondly the data already being stored and setup in the correct buffers, and requiring very little work performed in the way of converting types. The results of this test can be seen in Figure 4. When the same operation was done on the Nexus 7 the JSON files took almost 80,000ms to load and setup while the Binary files would take just over 40,000 ms to load and setup. These results show that by using the binary file over the JSON one, we are able to have improvements of around 50% across all devices.

The next test was to see how long it took to render a object, as we wanted to compare the speed up by not needing to switch between multiple buffers. To test this first we rendered 10,000 objects, with no texture or normals, and found that both of them take the same time, as the same number of buffers are used in each the binary file method and the JSON method at this point. By adding additional requirements, such as textures and lighting based on normals, this makes the rendering application need to switch between multiple buffers for the JSON file, but not for a binary one. With this test we found that once again that the binary file, and using an offset within a vbo proved to be a more effective method of processing. Figure 5 clearly shows that these results are true across all devices tested on. When tested on the Mac Pro the improvement was found to be 9.3%, as the JSON implementation took 75ms to complete while the binary one took a total of 68ms. The Nexus 7 the improvement was found to be 8.7%, with a time of 571 from the JSON file, to 519ms with the binary one. Finally the Samsung Galaxy S3 showed an improvement of 8.0%, with the binary file taking only 1928ms compared to the JSON one of 2096 ms.

For the last test, we compared the performance of the JSON and binary implementation again, but with also using normals, as to apply a lighting effect to the model, as seen in figure 3. This resulted as expected with an increase in time taken the reader the scene, caused by the increased requirements of the shader, along with the increase in data required to be passed. The improvement in performance of the binary implementation also increased over the JSON method, more than was tested previously. Once again Figure 5 shows the difference between the two methods, with gap between the two increased even more so. One the Mac Pro, the improvement was found to have become 12.2% with binary taking 72ms, down from the JSON one taking 82ms. On the Nexus 7 a similar improvement was found, at 10.5% or a 861ms time using JSON files, to 772 using the binary formate. With the Samsung Galaxy S3 the improvement was 9.6%, with the timing of the JSON file taking 3042, taken down to 2748 with the binary one.

Overall it can be seen that by utilizing a binary file to store the required information, along with using a single vbo with offsets to represent where each point begins and ends it is possiable to increase performance of graphical applications built on web based technologies.

4 Discussion

To create these benchmarks, an application was created which was able to render both models created using a JSON file, or one from a binary file. To ensure a fair test took place, the only differences were the setup and the rendering aspects of the application. Each of the two areas were tested separately as to ensure that the start up time did not affect the rendering timing.

The images below show the different effects applied to the teapot throughout the different stages of testing. Below in image Figure 2,is the effect of rendering the Utah teapot without any additional information. This implementation was the most simplistic as it used the least amount of information, and thus the least amount of processing by the graphics card. The implementation did not effect the performance, as the data which was passed, and how it was processed was the same for both binary and JSON files.

Figure 3 shows the uses of textures applied to the Utah teapot. In this implementation we see the first performance increase of the binary file and using offest within buffers. As shown in Figure 5 across all platforms tested, the binary was faster in rendering each frame than the JSON implementation, as it no longer is required to switch between buffers so rapidly. One thing to note is that the longer the renderings take to complete, the smaller the improvement was found to be.

Once the testing started to include the use of more

Figure 2: Utah Teapot, being rendered with not applied textured or lighting

Figure 3: Utah Teapot, being rendered with an applied texture

information, it became clear that the binary file became far more effect because of the way in which it renders an object. In Figure 4 we see the application of the lightning based on the normals passed through to the shader. By using the binary and offset data we are able to minimize the need to switch between 3 buffers to the use of only one, this improvement would become more prevalent with the more information being passed at a time.

One area looked into to help improve testing was designing a Chrome plug-in to help in capturing detailed performance, this proved to be unnecessary, as the best way to measure performance was found to be using the console within Chrome, with could also be accessed easily on android based devices.

With testing it was found that based on the firmware running on the device would affect its implementation. With the Samsung Galaxy S3, some fireware would not allow for the use of high precision data types on the shader, meaning that to test on this device a medium precision would need to

Figure 4: Utah Teapot, being rendered with an applied texture and lighting

be used, and this would effect the rendering speed across all devices. This problem was found to only occur on some firmware so did not effect the testing when run on firmware.

5 Conclusion

In summary we have managed to show two method in which to improve performance of platform independent graphics using WebGL. By converting the file to binary, with the correct arrays setup within, we manage to have a file which loads faster and is ready to be rendered in almost half the time as a JSON file, and one quarter the time of an wavefont OBJ file. The other improvement measured was that of the rendering speed, by removing the need to switch buffers multiple times for each draw call, this lead to the overall improvement of around 10% depending on the device, and amount of information being passed.

We believe that these changes are easily implemented into most applications based on WebGL, and can lead to greatly improved performance of an application, in startup and run time.

Future work would be to convert a JSON file using multiple VBOS, into one which used a single VBO, to allow for the ease of loading in a JSON file, with the performance found with the binary method.

In summary, various mobile devices hold great promise as platforms for rendering complex quality graphical models even using platform independent software and client approaches. Appropriate use of data compression and JSON significantly improves teh performance however.

References

[1] Andersson, S., Goransson, J.: Virtual Texturing with WebGL. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden (2012)

[2] Anttonen, M., Salminen, A.: Building 3d webgl applications. Tech. Rep. Report 16, Tamperer University of Technology, Finland, Department of Software Systems (2011)

[3] Bourke, P.: 3d stereo rendering using opengl (and glut) (2002), website PDF

[4] Chen, B., Xu, Z.: A framework for browser-based multiplayer online games using webgl and websocket. In: Proc. Int. Conf. on Multimedia Technology (ICMT). pp. 471–474. Hangzhou, China (26-28 July 2011)

[5] Congote, J., Segura, A., Kabongo, L., Moreno, A., Posada, J., Ruiz, O.: Interactive visualization of volumetric data with webgl in real-time. In: Proc. 16th Int. Conf. on 3D Web Technology (Web3D'11). pp. 137–145. Paris, France (20-22 June 2011)

[6] DeLillo, B.P.: Webglu development library for webgl. In: Proc. SIGGRAPH 2010. p. 1. Los Angeles, California, USA (25-29 July 2010)

[7] ecma international, Rue du Rhone 114, CH-1204 Geneva: ECMAScript Language Specification, 5.1 edn. (June 2011), standard ECMA-262

[8] amd Fritz Schneider, T.A.P.: JavaScript: the complete reference. McGraw-Hill (2012), iSBN 9780071741200

[9] Hearn, D., Baker, M.P.: Computer Graphics with OpenGL. No. ISBN 0-13-015390-7, Pearson Prentice Hall, third edition edn. (2004)

[10] Hill, F.S.: Computer Graphics Using OpenGL. No. ISBN 0-02-354856-8, Prentice Hall (2001)

[11] McMullen, T.H., Hawick, K.A., Preez, V.D., Pearce, B.: Graphics on web platforms for complex systems modelling and simulation. In: Proc. International Conference on Computer Graphics and Virtual Reality (CGVR'12). pp. 83–89. WorldComp, Las Vegas, USA (16-19 July 2012), cSTN-157

[12] Nixon, R.: PHP, MySQL & JavaScript. No. ISBN 978-0-596-15713-5, O'Reilly (2009)

[13] Preez, V.D., Pearce, B., Hawick, K.A., McMullen, T.H.: Human-computer interaction on touch screen tablets for highly interactive computational simulations. In: Proc. International Conference on Human-Computer Interaction. pp. 258–265. IASTED, Baltimore, USA. (14-16 May 2012)

[14] Woo, M., Neider, J., Davis, T., Shreiner, D.: OpenGL Programming Guide: The Official Guide to Learning OpenGL. Addison-Wesley, 3rd edition edn. (1999), iSBN:0201604582

[15] Wright, R.S., Haemel, N., Sellers, G., Lipchak, B.: OpenGL Superbible. No. ISBN 978-0-321-71261-5, Pearson, fifth edn. (2011)