EG UK Theory and Practice of Computer Graphics (2012)
Hamish Carr and Silvester Czanner (Editors)

Hardware Accelerated Medical Data Visualisation on the Web

H. Wei, G.J. Clapworthy, E. Liu, Y. Zhao and N.M.B. McFarlane

Dept. of Computer Science and Technology, University of Bedfordshire, Luton, UK

Abstract

This paper proposes a way to develop high-quality real-time web-based medical 3D data visualisation. Isosurface extraction is used as an example to discuss how to use programmable Graphics Processing Units (GPUs) and shaders to improve rendering performance on the web. The method is designed to reduce data transmission. When data is ready, the performance penalty can be considered negligible. A method to estimate memory usage to balance client memory limitation and rendering quality is also described. A way of using the frame rate to measure performance on the web is suggested, which could be used in future web visualisation.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics;

1. Introduction

Web applications always have to meet the challenges of speed of interactive response and computation, and this is particularly true in medical visualisation. Bandwidth remains a bottle-neck for the processing of large datasets on the web, and the transfer of data between the client and server is a major overhead of web applications. The increasing size of medical datasets, often a result of the increased resolution of medical imaging devices, means that this remains a problem in web-based medical applications. The need for high-quality fast response and recent developments in hardware have meant that, in the architecture of web applications, the roles of client and server have been subject to recent change.

Isosurface rendering is a good example - using volume data from a CT or MRI device, the isosurface formed from points within the data which have the same scalar value (the isovalue) is extracted. An isovalue represents regions of a particular tissue type and the volume can be viewed as a series of extracted isosurfaces, each representing the 3D volume bounded by a specific isovalue. Marching cubes [LC87] is probably the most popular isosurfacing algorithm.

During data exploration, each time the user changes the isovalue, a new isosurface is generated, and this should appear at interactive rates.

For a web visualisation system, pure client computation may be limited by hardware capability but a fat client will spend much time loading the program from the server.

Interactive manipulation and high-quality rendering are important in the medical area. Our system is designed to take advantage of a high-end server while achieving fast response at the client. To achieve this, we reduce transmission dataset between the client and server and improve the ability of the web client to process the data.

To reduce data transmission, both client and server are provided with a rendering module, so there is no need to exchange intermediate results between them - only data on the parameters necessary for the computation are transmitted. Data processing at the client is used for interactive viewing of the data and providing a fast response to the user. Only when further operations have to be performed on the surface using algorithms from an extension library (which, therefore, cannot be performed on the client), do we use the server for the processing. In such a case, we do need to exchange the dataset. This will be explained further in Section 3.

To improve web client performance, we use the graphics processing unit (GPU); it plays an important role in improving speed in computation-intensive tasks.

This paper presents a practical approach to the design of a web system for isosurface extraction and display using two methods - a hardware acceleration method and an estimated load based CPU method. Both support interactive visual ex-

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG12/069-076

ploration. We compare the performance with that of a stand-alone system and as a client by using the same hardware environment, algorithm, data and parameters for each. we also compare our method with webGL for development aspects and speed.

We summarize our contributions as follows.

- **a** new way of using GPU on the web that is different from, and faster than, current techniques based on webGL.
- **u**se of visualisation resources from both client and server to reduce data transmission.
- **e**stimation of the best Level of Detail for a resource-limited web client, which balances resolution and memory.
- **u**se of frame rate as a measure of performance on the web.

The remainder of the paper is organised as follows: Section 2 introduces recent related work from 3 aspects of visualising 3D objects over the web. Section 3 describes our method of rendering 3D medical data on the web and Section 4 presents the details of our approach using GPUs, illustrated by an example on isosurfacing. In Section 5 we analyse the performance of our work and in Section 6, we present future research directions.

2. Related work

In recent years, advances have been made in representing 3D objects over the web. In medical applications, the user generally needs to operate on their data interactively in real-time, as they would on a stand-alone system. We view the work that has been done previously from three perspectives: technical, architectures and algorithms.

2.1. Web technical

The Virtual Reality Markup Language (VRML) was a widely used file format for representing 3D interactive vector graphics. With the requirements of high image quality and performance speed, it has now evolved to the more mature and refined X3D standard. X3D [X3D] is a royalty-free open standard file format and run-time architecture to represent, and communicate among, 3D scenes and objects using XML. In 2003, Parisi [Par03] introduced the Flux player which implemented the X3D standard, and in 2004, an X3D component supporting programmable shaders was introduced for use in programmable graphics hardware, synthesising the X3D appearance model and shader effects [CGP04]. This showed that it was possible to achieve advanced rendering effects in X3D in real time, with interactivity.

The WebGL standard [CSK*11] brings hardware-accelerated, plugin-free 3D graphics to the web. It is a cross-platform, royalty-free web standard for a low-level, shader-based, 3D graphics API based on OpenGL ES 2.0. The difference from other technologies is that WebGL is an API not a plug-in, which enables direct access to 3D from the web page itself.

The Khronos group, which developed WebGL says "Developers familiar with OpenGL ES 2.0 will recognise WebGL as a shader-based API, with constructs in JavaScript that are semantically similar to those of the underlying the OpenGL ES 2.0 API", [Khr]. Since WebGL offers multi-browser support (Safari, Chrome, IE, Firefox), the Khronos group believes that the browser vendors will improve javascript performance sufficiently for a wide range of applications to use WebGL.

2.2. Architectures

In recent years, 3D Web application architectures have evolved gradually from a pure and simple web page, to a plug-in web page to a browser-supported API web page. 3D rendering has moved from remote (server) to local (client). One reason for this has been to gain interactive performance, another is the improvements in hardware to support the technology. Using a PC with a GPU as client is now viable.

In the early stages, a visualisation system processed the data, rendering it off screen on the server side, with only the resulting image being sent to the client. The user benefited from a lightweight client and a powerful server. The drawbacks were the delay in response from the server and the inability to manipulate the 2D image produced.

Later, for interactive manipulation, browsers could show objects in a 3D view with the help of a vrml/X3D plug-in, or other plug-in such as java 3D or java view, which was a great step forward. Todd and Jain [TJ95] developed web-based interactive 3D visualisation of biomolecular structures. using programmable shaders supported by X3D to create real-time results of high visual quality. To use X3D, they converted molecular structure data from the Chemical Markup Language *CML* to X3D and added pre-defined shaders as nodes into X3D during the translation.

[CDMG03], [EGE98] were early efforts in isosurfacing for volumetric data in a web-based system. The geometry defined by the iso value was computed on the server then loaded into a VRML-based browser. The drawback was the amount of data transmission required. Due to the limited network bandwidth, compressed or decimated data [SZ92] were often transferred instead of the original results, which could reduce the resolution of the output, though progressive refinement could be used to gradually improve the results displayed.

Local computation takes advantage of fast response times, reduced data transmission and reduced server-side pressure when client numbers increase. [CSK*11] presents a direct volume rendering *DVR* system for the web, which performs much of the rendering on the client machine using hardware-accelerated volume ray-casting implemented in webGL. Most of the computation is performed in vertex and fragment shaders written in GLSL which are run natively on the GPU hardware. WebGL is a javascript binding of the OpenGL ES 2.0 API and the performance of javascript interpreters was described as "not far from that of natively compiled languages". Their algorithm was designed to run entirely on the client and avoid the use of dynamic server content. However, client computation power is generally lower than that of the server, and if the whole burden is placed on the client, the computation may suffer. We describe our web client performance as close to that of a natively compiled Stand-alone program(see Section 5).

Disadvantages of webGL-based web visualisation centre on 3 key points. First, as WebGL is a javascript binding of the OpenGL ES 2.0 API, part of algorithm was written in javascript. Although javascript could be run on the server side, the goal of server-side javascript (SSJS) is to eliminate the gap between the client and server. However, it is seldom used for intensive computation. If we use it for a multi-function web application system, we should consider using another language to develop the server-side algorithm. From this point of view, the method we have developed is more suitable for taking advantage of a high-end server. Second, javascript runs as an interpreted language. Compared with a compiled language which is converted into machine code and then 'directly' executed by the host CPU, interpreting delivers a much slower speed of program execution on the host CPU. Finally, debugging code is not convenient, especially for complex algorithms with a large data set.

2.3. Algorithms

Isosurfacing is in common use for visualising volume data and there have been many examples of delivering isosurfaces over the web, often using a multiresolution approach to reduce the number of primitives transferred. In [KW03], isosurfaces were generated on the server side and, to reduce the number of polygons to be transmitted to the client, a progressive isosurface algorithm was used in which surface hierarchies were generated from the volume data with a level-of-detail (LOD) control. The user was able to reconstruct the surface at a coarse level of resolution and refine it, if required. In [CCD*99], [KDCS99], a similar approach, developed independently, used the coarse level as a preview, with progressive refinement automatically improving the output whenever the user was not interacting with it.

[EWE99] allowed the user to manually select a region of interest in which the surface was reconstructed at the finest level, with the surface outside that region being reconstructed at increasingly coarser resolutions.

The continuing increase in volumetric dataset size may prohibit the handling of these models on affordable low-end single-processor architectures,and surface-based techniques may not be suitable for visualising volumes that contain many isosurfaces that are important to the user, because extracting all of the interesting surfaces would take too much storage to be practical [GDL*02]. DVR techniques [LCD09], [GS03] are popular as their high performance allows easy switching of isovalues. We created a DVR method on the web in which the rendering was run directly on the front end to gain speed.

The challenge for a web system is how to use existing resources to achieve fast response and fast rendering, while at the same time providing access to suitably rich algorithmic resources.

3. The proposed software framework

Nowadays, the increasing power of graphic cards allows fast calculations [KW03] and, in isosurface rendering, the user will expect to be able to change the isovalue interactively.Thus, the new isosurface must be generated rapidly at the client to save transmission time. Web-based techniques such as asp, php, jsp, run on the server side and use server-side resources, rather than client resources. A java applet run locally can use local resources file system, CPU, GPU, RAM, once it has been verified.

However most graphics algorithms are written in C/C++ or are based on such libraries. The Visualization Toolkit (VTK) can act as a bridge between graphics algorithms and web application as it supplies a java wrapper layer so it could be used via a java applet. VTK also supports shaders written in both GLSL and CG, so it provides a convenient method for loading GLSL shaders.

We have designed a multi-function web system for the interactive visualisation of 3D medical data and have used isosurfacing to verify the design. Our architecture takes advantage of both the client and server sides.

In Figure 1, the same rendering module is deployed on both the client and server and is run in two forms. On the server side, it is run as a library, while on the client side, it is run in a java applet as a plug-in embedded in a web page.

The benefit from this approach is that the client has computational ability and thus does not need to ask for results from the server in every case. VTK provides the main visualisation tool in the rendering module. It supplies not only a view on the client but also algorithms that can be easily extended. This gives the client flexibility in data processing, while the java wrapper acting as an upper layer in this module supplies an interface for exchanging parameter data between the client and server.

On the server side, we deploy the same render module, so it is easy to recover intermediate results generated through the client at the server by transferring only the relevant parameters. The difference between the client and server in data processing is that the server can integrate other extended

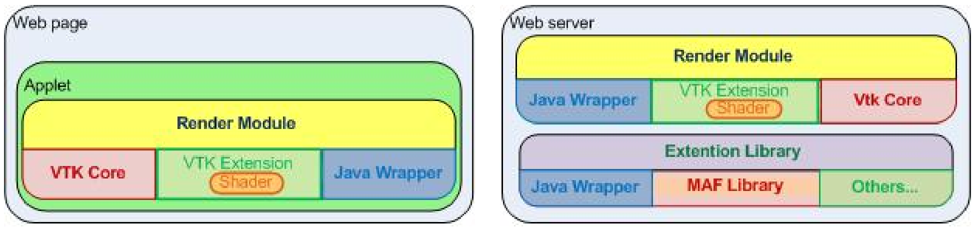

Figure 1: *System deployment.*

libraries based on different techniques to enlarge the algorithms and gain server computation. We provided a Multimod Application Framework (MAF) [VZT*07] platform as an algorithm library at the server side. MAF is an open source framework containing several libraries such as VTK and ITK. The java wrapper layer is its interface with java-based server applications.

As the render module has a java wrapper layer, it can be invoked by a java applet at the client or by a servlet (a small java program that runs within a web server). An applet can communicate with a servlet and can also visit an element value on the web page or a javascript method through the JSObject class. A javascript method can also visit a public attribute or invoke a public method of an applet. In this way, communication was constructed between the web client and the server.

As it is a multi-function system, the data output from one operation might act as input for the next. Let us again take isosurfacing as an example. The user extracts several surfaces at the client, views them, then selects one of them on which to perform further operations such as smoothing, filling holes, deformation, etc. These operations might need support from MAF at the server side, so the surface geometry is input for these algorithms. To avoid having to transmit it, the geometry is generated on the server side. The parameter for extracting the surface from the volume data is a scalar isovalue. The new operation might need other parameters such as a selection area from the surface which can be performed by a VTK plane widget. These parameters are delivered to the server side, using the isovalue to extract the same surface as the client did and export its geometry as input for a MAF algorithm. After processing, the data generated from the MAF algorithm is delivered to the client for interactive viewing of the result.

Our render module is aimed at rendering data and performing processing related to user interaction. Its algorithms come from vtk core and from libraries extended from vtk. The other richer algorithmic resources come from other extension libraries such as MAF. We could not put such big libraries on a web client as they would cause delays for their loading, so complex operation are processed on server. It is also far easier to extend facilities by integrating additional libraries at the server side. As the server has the same Render Module, it does not need to ask for the extracted surface from the client. By using the same isovalue, the server will extract the same surface as client did.

The benefit from having the same render module on both client and server is that it can balance the computational burden and reduce the need for transmission of data. In the example above, there is no need to pass the surface data between the client and server. Also, as the render module uses the same code at both the client and server it needs to be developed only once - there is no need to develop them twice with different languages.

4. Rendering

In isosurface extraction there are two goals. One is to render it to the screen, which needs a fast algorithm. The other is to find the geometry of the surface, so that it can be exported or transmitted for other views or for the server side to perform further operations. The surface method requires a preprocessing step before rendering to build the geometric structure, while DVR can produce good quality with high speed, with the help of a graphics card.

To achieve the two goals, we performed two different applications, one on the CPU, the other on the GPU. Although the GPU provides high computational speed, there may still be users whose machine is without a suitable GPU. For these users, the CPU method is more suitable and it has other advantages – it uses multiresolution LOD rendering, so users can select coarser levels to review the whole dataset and they can also export the geometry of the extracted surface for further operations.

Our work uses VTK, which divides all the classes and their wrapper classes into different function libraries and their wrapper libraries. When we use specific classes, we load only the corresponding wrapper libraries into the JVM at runtime. The JVM searches for these libraries in the "java library path" in local. The wrapper libraries are not independent, however; they rely on the corresponding function libraries when they are executed. To run the code, the necessary function libraries and their wrapper libraries need to be deployed in the machine on which they execute.

4.1. Using GPU

To take advantage of the advanced features of recent graphics cards, VTK supports the incorporation of programmable hardware shader technologies. We implemented the method of Liu et al. [LCD09] and applied it on the web. All the codes created are in a pure VTK project. As VTK is a high-level tool, based on OpenGL, we can still use OpenGL easily in a VTK project. VTK supplies the user with the mechanics of loading, compiling, binding, setting variables for a shader, so the developer can focus on shader development. If an error arises from a shader, VTK will report the error through its error macros and default to use the OpenGL pipeline. In this way, the connection between the web page and the hardware card was built: web page $\Longrightarrow$ java applet $\Longrightarrow$ VTK $\Longrightarrow$ OpenGL $\Longrightarrow$ shader $\Longrightarrow$ GPU .

For the GPU algorithm, we use the cell rasterisation [LCD09] for "fast, high-quality, GPU-based" isosurface rendering. This method produces smoother images than both Marching Cubes and the normal raycasting technique by reducing the diamond artefacts. The main idea of the algorithm is to extract and render all the active cells instead of extracting and rendering all of the marching cube triangles individually.

This method improves performance in the following ways. Firstly, it uses the HistoPyramid texture [ZTTS06] to extract active cells and skip all the inactive cells from the coarse volume data; this produces considerable savings as the active cells generally occupy only a small percentage of the total volume. Secondly, it treats the extracted active cells as rendering primitives instead of preprocessing the data to build a accelerating structure for a particular isovalue; this means that one can easily switch between isovalues. Thirdly, a CPU sorting algorithm is used to sort the sequence of extracted cells from a quad-tree order into a view-dependent order so that the hardware early-z-culling feature can be applied, which means that the hardware program flow focuses only on the front-most cells and avoids rendering occluded cells.

We begin the algorithm from an inherited abstract class, volume mapper: vtkvolumeMapper, which can be used for rendering geometry or rendering volumetric data. At the first stage, the method extracts the active cells from volumetric data by comparing the scalar values associated with the 8 corners of each texel with a specific isovalue to build a HistoPyramid, retaining the active cells. The isovalue as threshold is passed to the fragment shader.

One OpenGL program object (which represents a useable part of the render pipeline) was used to build the base level of this quad-tree. Another program object was used to generating the next pyramid level - the GPU repeatedly sums four adjacent cells, each time halving the resolution until only one cell remains. Now the HistoPyramid is built, all the active cells in a list can be retrieved by traversing the HistoPyramid in quad-tree order. If the isovalue does not change, there is no need to rebuild the HistoPyramid during interactivity, as the active cells do not change. The second stage is to render each extracted active cell as a point primitive. The vertex shader retrieves the object-space position of the cell and projects it in screen-space. Phong shading is performed for the fragment shader.

We developed this program in a C++ environment. For the web program, the libraries and shader files are deployed on the client machine automatically. The shader source code must be set in shader objects when the GPU algorithm is invoked. They are stored in the same folder as the VTK libraries.

As client needs a java runtime environment to run a java applet, it must have a JRE folder in the "java library path". The JRE is deployed as a plug-in which the user can use to install it before running the main program in case JRE was not previously installed. The JRE execute folder is the folder in which we deploy our libraries and shaders. When it is executed on the client by an applet, the shader source files are downloaded and deployed automatically.

4.2. CPU algorithm

Users with a low-end computer as the client will use the CPU algorithm – we provide adjustable resolution levels to control the number of primitives used to represent the surface. VTK has a *vtkMachineCube* class to generate isosurface output from volume input. However, we did not use this directly, but instead used a method similar to Marching Cubes. The extracted triangles are cached for use while the isovalue remains unchanged and an LOD control is provided – using low resolution improves the speed for large datasets. Further, a function is supplied for automatically computing the best LOD for the data.

Since the rendering is performed at the client, some code is in the form of a java wrapper, and some code is in the form of dynamic libraries in native code which are written in C++. Every client has limited random-access memory (RAM), to avoid a jvm "out of memory" error from java wrapper or "could not allocate memory" error from the native code. We supply the best LOD function for user. Although developers are allowed to specify a larger-than-default maximum heap size for an applet via the Java_arguments parameter, we should not allocate too much memory for JVM. The JVM heap and stack stores all objects and method invocations and the local variables created by the java codes, however the native VTK library code uses memory from native memory. Java runtime is itself a native program that consumes native memory, which means the client machine's hardware and operating system (OS) impose the limitations on native memory. To avoid these memory errors, we use a mechanism to estimate how much memory the dataset will use at a specific LOD level. This enable the user to select the highest LOD for rendering that will provide a suitable usage of memory and produce an acceptable frame rate.

The storage of a triangle is easy to compute (three vertices and normals), and the estimate is based on the assumption that the more triangles that are used to represent the surface, the more memory is needed.

The volume data is divided into fixed size (8*8*8) blocks (see Figure 2); the min and max isovalues for each block are calculated by looping through all voxels. For a given isovalue, the number of relevant contour blocks can be calculated – only these contribute to the final surface. We divide the LOD into 4 levels (0,1,2,3) in which a cube size could be (8^3,4^3,2^3,1^3),a block could contain 1 cube for level 0, 2^3 for level 1, $4^3$for level 2 and 8^3 for level 3. The estimated number of triangles for the surface is Num_triangles_per_cube * Num_cubes_per_block * Num_relevant_blocks. The gradient of the original data is used for shading the models.

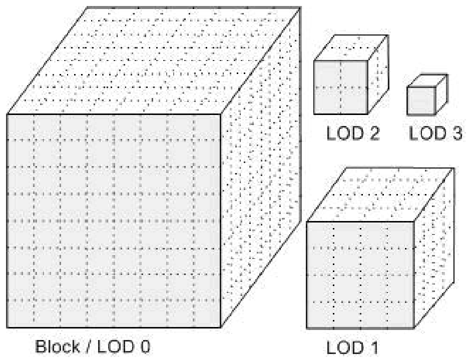

Figure 2: *Levels of detail.*

For an isosurface with a given isovalue and LOD, there are three cases when estimating the number of triangles. If it has been computed previously, there is no need to compute it again, as an array stores all of the computed numbers of triangles for different LODs at a specific isovalue. If, for this isovalue, we have computed numbers for other LODs, our estimate is based on the triangle numbers at higher resolutions. As the cube volume is inversely proportional to the precision (number of triangles), it is easy to compute. If this is the first calculation after a change of isovalue, we use an empirical constant value: the number of triangles in every cube in a block containing the contour is set to 2. If this is too high, the rendering may be produced at a lower resolution than optimal; if it is too low, the rendering might take a long time or be limited by RAM.

Figure 3 shows detail from images at different LODs. The data used is a 68M heart volume data set - Figure 3 shows only part of the data. Using the same dataset, the resolution affects the final rendering quality. The dimension of this dataset is 512*512*512. Table 1 shows the numbers of triangles for the 4 LODs. From Figure 3 and Table 1, it is clear that the more triangles that are present, the smoother the surface.

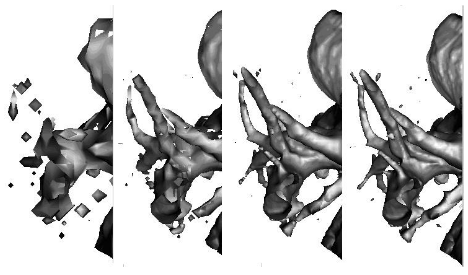

Figure 3: *Image close-ups at increasing LOD.*

Table 1: *Numbers of triangles at different LOD.*

Level 0	Level 1	Level 2	Level 3
19,896	159,170	1,273,363	10,186,907

5. Performance Analysis

The results in Figure3 were rendered by web GPU from a raw dataset with size 68MB. The program run on a PC with an Nvidia GeForce 8800GTX graphics card, an AMD 3.0GHz processor and 3.25GB RAM.

As the speed of a web program is limited by bandwidth, we cannot effectively measure the computational speed of a web program compared to a stand-alone program performing the same task. However, we measured it when the code is already downloaded to the client and the data are ready. Now, its running speed can be compared with the stand-alone program. The frame rate will change when the object on the screen is viewed from a different position or at a different angle. The more pixels on the screen, the more parallel threads are needed.

Hence, we set the dataset at a fixed angle to the camera (see top row of Figure 4) and enlarge the rear data boundary to 18.2cm width and 9.0cm height on the screen. The screen resolution is 1920*1080 pixels. At a key press, the active camera is rotated automatically through a full circle, one degree a time, creating 360 renderings. We performed the same measurements 10 times on both the stand-alone program and the web program. We also tested the latter on Firefox 8, IE 8 and Chrome 15.0. The results are presented in Table 2.

Table 2: *Frame rates on different browsers.*

Environment	Stand-alone	Chrome	IE	Firefox
Time(secs)	11.941	12.303	12.324	12.646
fps	30.15	29.26	29.21	28.43

Chrome performs best amongst the browsers, with a frame rate of 29.26 fps. That means there is a 0.001 [(12.303-11.941)/360] second difference for one rendering between the best web program and the stand-alone program.

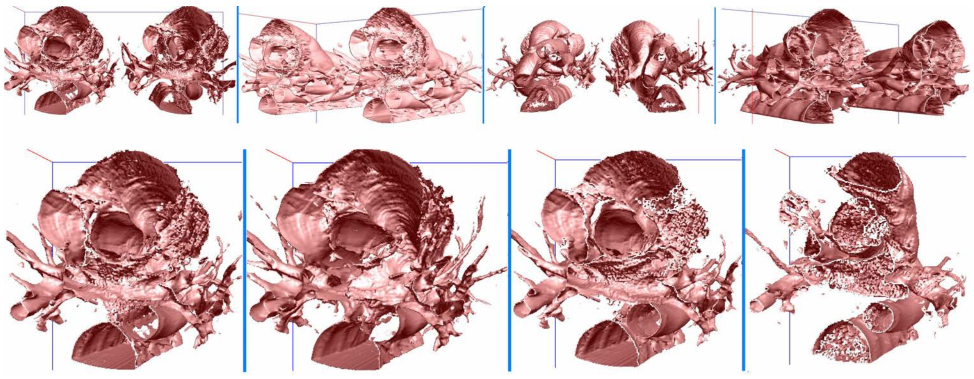

Figure 4: *Results from the GPU algorithm.*

This is because, with our web client program using VTK java wrap to drive the VTK core program, the VTK core program invokes the OpenGL interface to realise the GPU rendering. But the stand-alone program uses the VTK core program directly, so there are a little performance loss in the JNI interface.

Another test on this data to compare the Stand-alone and web programs is to extract an isosurface 95 times as a group for different isovalues (see bottom row of Figure 4) - we used equidistant values in the range [0,65535]. Each time the isovalue changed, the program had to re-compute the surface. This was done 10 times. The stand-alone program needed 0.0693 seconds for one isosurface extraction and rendering, while the web program needed 0.0721 seconds. The difference between them was 0.00276 seconds.

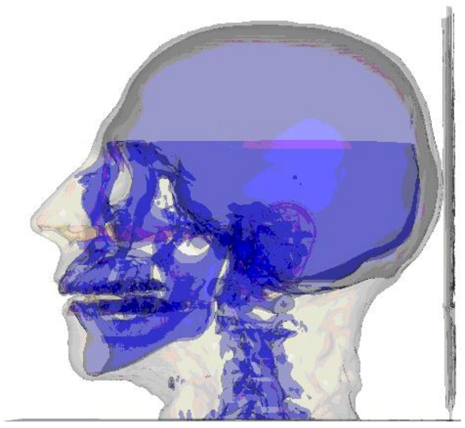

Figure 5: *Multi-layer rendering.*

Figure 5 shows bone and skin layers extracted from volume data using the GPU method. As noted in [LCD09], this is capable of rendering multiple layers with different isovalues in a single pass. As a result, the rendering of multiple transparent isosurfaces can be achieved at no extra cost.

6. Future work

The DVR approach of [LCD09] is much more suitable for processing dynamic volume data and we intend to investigate this in the future.

Our method for estimating memory usage could be extended to include an estimate of the computational time; both estimates could then be used to balance better the compute burden between the client and the server.

The main idea – fast response created by using local resource(GPU, RAM) through native code - can be used on other web systems and other application areas.

References

[CCD*99] Crudele M., Clapworthy G. J., Dong F., Krokos M., Salcito G., Vasilonikolidakis N.: Accessing a WWW reference library of 3D models of pathological organs to support medical education. In *Proc Medical Infor-matics Europe 99* (1999).

[CDMG03] Clematis A., D D'Agostino, Marco W. D., Gianuzzi V.: A web-based isosurface extraction system for heterogeneous clients. In *Proc. 29th Euromicro Conference* (2003), pp. 148–156.

[CGP04] Carvalho G., Gill T., Parisi T.: X3D programmable shaders. In *Proc. 9th International Conference on 3D Web Technology* (2004), pp. 99–108.

[CSK*11] 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 International Conference on 3D Web Technology* (2011), pp. 137–146.

[EGE98] Engel K., Grosso R., Ertl T.: Progressive isosurfaces on the web, Late Breaking Hot Topics. In *IEEE Visualization* (1998).

[EWE99] Engel K., Westermann R., Ertl T.: Isosurface extraction techniques for web-based volume visualization. In *Proc. IEEE Visualization '99* (1999), pp. 139–146.

[GDL*02] Gregorski B., Duchaineau M., Lindstrom P., Pascucci V., Joy K.: Interactive view-dependent rendering of large isosurfaces. In *Proc. IEEE Visualization '02 (VIS 02)* (2002), pp. 475–484.

[GS03] Gao J., Shen H.: Hardware-assisted view-dependent isosurface extraction using spherical partition. In *Proc. Symp. on Data Visualisation* (2003), pp. 267–276.

[KDCS99] Krokos M., Dong F., Clapworthy G. J., Shi J. Y.: Towards fast volume visualisation on the WWW. In *Proc Information Visualisation '99* (1999), pp. 286–291.

[Khr] http://http://www.khronos.org/webgl.

[KW03] Kruger J., Westermann R.: Acceleration techniques for GPU-based volume rendering. In *Proc. IEEE Visualization* (2003), pp. 287–292.

[LC87] Lorensen W., Cline H.: Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics Forum 21*, 4 (1987), 163–169.

[LCD09] Liu B., Clapworthy G. J., Dong F.: Fast isosurface rendering on a GPU by cell rasterization. *Computer Graphics Forum 28*, 8 (2009), 2151–2164.

[Par03] Parisi T.: FLUX: Lightweight web graphics in XML. In *ACM SIGGRAPH 2003 Web Graphics Presentation* (2003).

[SZ92] Schroeder W. J., Zarge A.: Decimation of triangle meshes. *Computer Graphics Forum 26*, 2 (1992), 65–70.

[TJ95] Todd T., Jain E. R.: Web-based volumetric data retrieval. In *Proc. 1st Symposium on Virtual Reality Modeling Language* (1995), pp. 7–12.

[VZT*07] Viceconti M., Zannoni C., Testi D., Petrone M., Perticoni S., Quadrani P., Taddei F., Imboden S., Clapworthy G. J.: The Multimod Application Framework: a rapid application development tool for computer aided medicine. In *Comput. Methods Programs Biomed* (2007), vol. 85, pp. 138–151.

[X3D] X3D specifications. http://www.web3d.org/x3d/specifications/.

[ZTTS06] Ziegler G., Tevs A., Theobalt C., Seidel H.: On-the-fly point clouds through histogram pyramids. In *Proc. Vision, Modelling and Visualization* (2006), pp. 137–144.