Mobile Visualization of Biomedical Volume Datasets

Movania Muhammad Mobeen, Lin Feng,
Nanyang Technological University, Singapore

***Abstract*— The WebGL platform has been introduced based on the OpenGL ES 2.0 API. It allows scripts embedded in a web browser to have native access to GPU hardware. Now that more and more real-time systems are moving towards a cloud-based architecture, it becomes important to capitalize on existing tools to extend the biomedical imaging and visualization domain. One such tool that can enable ubiquitous biomedical imaging and visualization is the WebGL platform. Existing work relies on a multi-pass strategy. We extend the visualization using a single pass approach. This gives much better performance especially on the mobile platforms where every additional texture access is costly. Quantitative evaluation reveals that the proposed algorithm outperforms the existing algorithm by a consistent 2x speedup not only on desktop platforms but also on the mobile platforms. Current mobile phones and tablets have limited support for dynamic loops thus, sampling rate cannot be changed dynamically and high quality renderings cannot be carried out. To circumvent these problems, we present the first 3D texture slicer. Since 3D texture slicing uses the rasterization hardware and support of the rasterizer is pervasive, we can not only modify the sampling rate but also carry out advanced effects. The design of our approach and extensive experiments are presented in this paper which proves the effectiveness of the proposed approach for pervasive biomedical data processing and visualization.**

***Index Terms*— Ubiquitous computing, Biomedical imaging, Data visualization, Biomedical image processing, Computer graphics**

I. Introduction

THE healthcare scenario has changed to accommodate better clinical decision and higher patient satisfaction. It has moved from traditional one-point contact to a conglomeration of multidisciplinary people working together to obtain best results. With the availability of internet, we have seen a large number of hospitals using integrated Health Information Systems (HIS) which help them to maintain a seamless flow of patient's information, insurance, clinical data, etc. between different departments. However, currently these systems are often synonymous with filling innumerable forms and storing bulky files filled with patient information. In the instrumental examinations, such as Computed Tomography (CT) and Magnetic Resonance Imaging (MRI), the resulting images and documents have to be processed offline. Ambiguous and incomplete data, or data fragmentation, often lead to lack of overview, and these drawbacks may impede the continuity and quality of care. Moreover, these information systems are limited to the niche of a hospital or its subsidiaries.

In a web-based system, the medical information and tools available on an integrated platform can be accessed by different people at any geographical location at any point of time. There are various portals which handle information transfer and storage but little work has been done on data processing. Online availability of biomedical image processing and visualization would improve accessibility of remotely located tools and also maintain congruency of processes applied by different users.

Biomedical image visualization is based on identification of appropriate features to assist practitioners to differentiate one type of objects, e.g. tumorous tissues, from another, according to their morphological characteristics. Challenges to graphics researchers are mainly in supporting standard and accurate assessment of images using combined set of filters present at one location. Currently, sophisticated volume rendering and feature visualization programs are only available on a stand-alone workstation. Data processing capability on a mobile device such as a smart phone is very limited.

In this connection, WebGL is proposed as a new standard for plugin-less high quality high performance graphics in a web browser. It is a cross-platform, immediate mode royalty-free web standard for a low-level 3D graphics API. Since WebGL is based on the OpenGL ES 2.0 API which is a subset of OpenGL for embedded devices, it uses the same shader language framework as the desktop OpenGL. Before WebGL, most of the 3D content was available only through plugins for the web browsers which often had compatibility issues and were not a write-once-run-everywhere solution. Moreover, such plugins had to be manually installed before any 3D content could be viewed. With WebGL, applications can now have native access to the graphics hardware through the well established OpenGL ES API without any plugin [1].

For application development, WebGL is exposed through the HTML5 canvas element as a collection of Document

Manuscript received September 16, 2012. This work is partially supported by two research grants, M408020000 from Nanyang Technological University and M4080634.B40 from Institute for Media Innovation, NTU, and a grant MOE2011-T2-2-037 from Ministry of Education, Singapore.

Movania Muhammad Mobeen is with the School of Computer Engineering, Nanyang Technological University, Singapore, e-mail: mova0002@ e.ntu.edu.sg.

Lin Feng is with the School of Computer Engineering, Nanyang Technological University, Singapore, e-mail: asflin@ntu.edu.sg.

Object Model (DOM) interfaces. It is entirely shader-based API bringing 3D to the web, and implemented right into the browser. Major browser vendors such as Apple (Safari), Google (Chrome), Mozilla (Firefox), and Opera (Opera) are members of the WebGL Working Group. WebGL uses the low-level javascript API to gain access to the power of a mobile device's graphics hardware such as GPU from within scripts on web pages. It makes it possible to create 3D graphics that update in real-time, run in the browser, and can also be run in any OpenGL ES 2.0 compliant mobile device. Currently, it is available in a number of smart phones and tablet computers from numerous vendors.

II. Previous Work

GPU-based direct volume rendering has been used for visualization of medical and scientific datasets. Numerous approaches have been proposed in the literature [2], [3], and for a more recent work, the SIGGRAPH course notes in [4]. Initial GPU-based approaches focused on using the fragment shader pipeline in a multi-pass approach [5] which renders front and back faces of a unit color cube. Then, rays are generated using textures rasterized in the first pass as lookups in a fragment shader. Thanks to its simplicity, this still remains as an effective approach for GPU ray casting.

With the introduction of loops in shaders in the Shader Model 3, a single-pass approach was pioneered by Stegmaier et al. [6]. Similar to other fragment shader based approaches, first a full screen quad is rendered on screen in order to invoke the fragment shader. Then, the ray casting fragment shader is applied. Using the assigned texture coordinates, the ray directions for sampling of volume are determined. Finally, the volume is traversed front-to-back. Such a single-pass approach shows great potential although more improvement is needed, especially for mobile devices where every additional texture lookup degrades performance considerably.

The versatility of GPU ray casting allows ray functions to be efficiently implemented with both front-to-back and back-to-front variants, and to be modified in real-time in the fragment shader. This helps to accommodate various ray functions like maximum intensity projection (MIP), maximum intensity difference accumulation (MIDA) [7], composite accumulation, psuedo-isosurface rendering [8] etc. in real-time without additional modifications.

The WebGL API was introduced only recently. In the medical domain, there has been some work using the new API, including the Google Body project [10]. However, there are very few reports on using WebGL for volume rendering. The only relevant work was reported in [9]. The authors presented an algorithm for extracting the 2D slice from the 2D flat texture layout. This approach uses the Kruger and Westermann's multi-pass approach [5] of rendering the front and back faces of a unit color cube in an offscreen buffer. Then, it uses these to generate the ray directions for sampling through the volume dataset in the fragment shader. The multi-pass approach for GPU ray casting requires additional textures in the first pass. The ray direction is calculated in the fragment shader in the second pass. For the mobile platforms, each additional texture fetch degrades the performance significantly. In addition, their method suffers from visible artifacts at the unit color cube edges due to rasterization. Moreover, it uses a fixed transfer function which is loaded from an image, which limits the flexibility in an interactive rendering session on a mobile device. This motivated the development of a single-pass approach for GPU ray casting in WebGL [13].

We present two volume rendering algorithms using the WebGL platform for implementing medical image visualization on the mobile devices. A remarkable advantage of our approach is that it can run directly on most embedded and mobile devices. In the next section, we will particularly introduce a novel single-pass rendering pipeline, in contrast to the current WebGL volume rendering systems which use the multi-pass algorithms. The new pipeline can be implemented efficiently on the embedded GPU in the mobile device, enabling real-time visualization of high-resolution volumetric datasets. We will also extend the pseudo-color shading by using customable transfer function widgets, which enables interactive feature enhancement in rendering.

Currently, the mobile devices and smart phones have limited support for shaders. Hence, variable length loops are limit to a compile time constant. This limits the volume rendering algorithms since, advance shading effects cannot be carried out. Moreover, the sampling rate cannot be adaptively modified without recompilation of the shaders. To circumvent these problems, we propose the first 3D texture slicer for WebGL. Since texture slicing relies on the rasterizer hardware and because the support for the rasterizer hardware is better on mobile platforms, we can not only carry out advanced shaders but also change the sampling rate at runtime. This enables us to carry out advanced shading [14] as detailed in the later section.

The rest of the paper is organized as follows. Section III details the development of our system on the WebGL platform which includes both the single pass ray caster and the 3D texture slicer. Section IV presents the experimental results and performance assessments we carried out on both desktop and mobile platforms. Finally, section V discusses and concludes this paper.

III. System Development

WebGL is an extension of the HTML5 canvas element. This element provides fast and high performance graphics constructs for rendering high quality graphics in a browser window. For 2D graphics, this is usually accomplished by using a 2D rendering context called the *CanvasRenderingContext2D*. For 3D rendering, the canvas element provides a rendering context, called the *WebGLRenderingContext* which provides the OpenGL ES 2.0 functionalities. Both of these API are controlled through javascript.

A. *WebGL supported system architecture*

Internally, the OpenGL context makes calls to the graphics hardware directly through the graphics driver (see Fig. 1). Therefore, there have been several debates on the security issues and vulnerabilities of the WebGL API. While such a low level access is necessary for high performance graphics, this also allows the client program to have access to the hardware directly which could potentially be used for an attack. Nevertheless, such security issues have been addressed by the new specifications and more security features are being introduced with each new specification update.

In the case of WebGL, a rendering context is obtained from the HTML5 canvas element. This is usually accomplished using javascript code. Most of the current WebGL implementations provide an experimental WebGL context which does not comply with the full WebGL specifications. In the new specifications v 1.0.1, however, the implementations are now required to provide a pure non-experimental WebGL context. The returned gl context can then be used to call any OpenGL ES 2.0 API function [1].

With the type-less javascript API, the object type and their memory management is handled by WebGL. This has some performance implications especially due to the strongly typed nature of OpenGL. WebGL has introduced special typed arrays that correspond to the native types used by OpenGL. This allows proper argument passing to the native OpenGL call from WebGL. With newer browser release, their javascript engine performance improves and becomes comparable to a native call as in C/C++ [1].

B. *Volume Rendering*

The following sections list the two volume rendering approaches implemented in our WebGL based volume rendering system. The framework of our WebGL based single pass volume renderers is shown in Fig. 2. The volume dataset is first stored into the GPU texture memory. Usually, this is carried out by using 3D textures. Since WebGL is based on the OpenGL ES 2.0 API which is a restricted subset of the desktop OpenGL, there are some functionalities missing, in particular, there is no 3D texture which is a crucial requirement for volume rendering. As shown in the figure, in our design, this limitation is circumvented by tiling each slice of the 3D texture into a 2D flat texture layout [9].

The volume rendering approach is based on an optical model to approximate the emission/absorption characteristics of the participating medium. The volume rendering integral models the emission, absorption and transmission properties that map the physical attributes stored in the 3D density function. The emission absorption model is given as

$$I(D) = F(s_0, D)I_0 + \int_{s_0}^{D} q(s)F(s, D)ds$$
$$F(a,b) = e^{-\int_a^b \kappa(t)dt} \quad (1)$$

where, I_0 is the background light intensity, F is the transparency function, $q(s)$ is the source term (the illumination model used), I is the intensity leaving the volume, $\kappa(t)$ is the extinction coefficient, D is the distance of the viewer from the volume and $\int\kappa(t)dt$ is the optical depth which gives the distance the light travels before it is absorbed. For a transparent material, the optical depth is small and for a more opaque material, it is large.

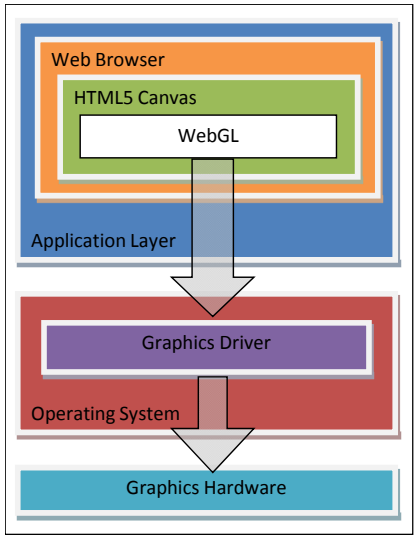

Figure 1. Hardware Design of WebGL

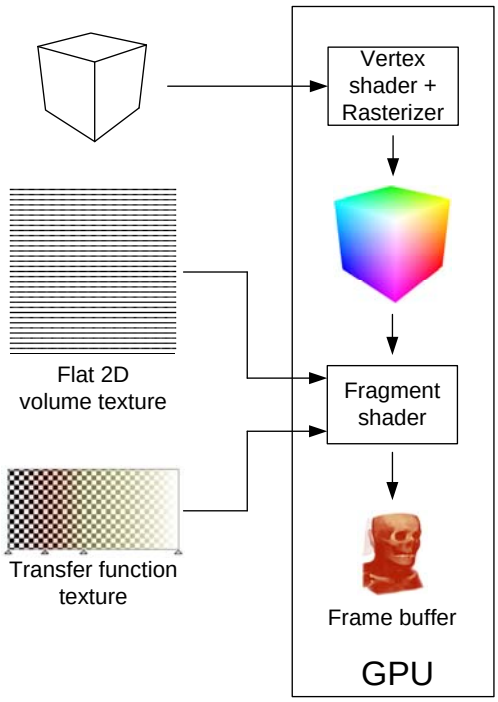

Figure 2. The dataflow for WebGL based volume renderer

Equation 1 only captures emission and absorption ignoring the scattering effects. Usually, single scattering is approximated using a local illumination model such as the Blinn Phong model. The normal for shading in this case is

estimated by evaluating the gradient at the current sample point using finite difference approximation. The volume rendering integral given in Eq. 1 cannot be solved analytically. Hence, numerical approximation is used by partitioning the integration domain into sub-intervals from s_0 to D such that $s_0 < s_1 < \ldots < s_{n-1} < D$. The integral is then approximated as,

$$I(s_i) = F(s_{i-1}, s_i) I(s_{i-1}) + \int_{s_{i-1}}^{s_i} q(s) F(s, s_i) ds$$

Replacing $F(s_{i-1},s_i)$ with T_i and the right side integral that is $\int q(s)F(s,s_i)ds$ with C_i, we get

$$I(s_i) = T_i I(s_{i-1}) + C_i$$

The total radiance at the exit point of volume is then given as

$$\begin{aligned} I(D) &= I(s_n) = T_n I(s_{n-1}) + C_n \\ &= T_n (T_{n-1} I(s_{n-2}) + C_{n-1}) + C_n \end{aligned}$$

This can be given with the following recursive expression

$$I(D) = \prod_{j=i+1}^{n} T_j I(s_{j-1}) \sum_{i=0}^{n} C_i, \quad with \quad C_0 = I(s_0)$$

In practice however, the transparency term (T_i) is usually replaced by opacity ($\alpha_i = 1 - T_i$).

1) Composite Rendering

The physical model for volume rendering relies on modeling the interaction between light and the participating media. The composite rendering tries to approximate the volume rendering integral through finite difference approximation. Typically, this mode is implemented through front-to-back or back-to-front alpha compositing. The discretized volume rendering integral is iteratively computed using either front-to-back or back-to-front compositing depending on the direction the volume is viewed from. When front-to-back compositing is used, the iterations are given as

$$\overline{C}_i = \overline{T}_{i+1} C_i + \overline{C}_{i+1}$$

$$\overline{T}_i = \overline{T}_{i+1}(1 - \alpha_i) \qquad with \quad \overline{C}_n = C_n, \overline{T}_n = 1 - \alpha_n$$

In the above equations, the terms C_i and α_i are obtained from the transfer function. Replacing the term C_{i+1} with C_{dst} and C_i with C_{src}, $\alpha_{src} = 1 - T_i$ and $\alpha_{dst} = 1 - T_{i+1}$, and reformulating, we get the conventional blending equation for front-to-back compositing

$$C_{dst} = C_{dst} + (1 - \alpha_{dst}) C_{src}$$

$$\alpha_{dst} = \alpha_{dst} + (1 - \alpha_{dst}) \alpha_{src}$$

For back-to-front compositing, the direction of traversal is reversed. Hence, the iterations are given as

$$\overline{C}_i = \overline{T}_i C_{i-1} + \overline{C}_i$$

$$\overline{T}_i = \overline{T}_{i-1}(1 - \alpha_i) \qquad with \quad \overline{C}_0 = C_0, \overline{T}_0 = 1 - \alpha_0$$

In this case, there is no need to explicitly store and calculate transparency. Replacing the term C_{i-1} with C_{dst} and C_i with C_{src}, $\alpha_{src} = 1 - T_i$ and $\alpha_{dst} = 1 - T_{i-1}$, and reformulating, we get the conventional blending equation for back-to-front compositing

$$C_{dst} = C_{src} + (1 - \alpha_{src}) C_{dst}$$

2) Single-pass GPU Ray Casting

When using GPU for ray casting, a unit cube is first rendered. Using the vertex shader, the clip space positions are estimated by multiplying the per-vertex positions with the modelview and projection matrices. During estimation of the clip space position, the vertex shader also outputs object space position for shading calculation. Following the vertex shader, using the connectivity information, the triangle is rasterized and the fragments are obtained. On each fragment, the fragment shader is invoked. Once the positions are obtained, rays are cast into the volume in the fragment shader. These rays are sampled and adjacent material values are used to reconstruct the original data, typically using trilinear interpolation.

In our single-pass GPU ray caster, the bulk of the work for volume rendering takes place in the fragment shader. The shader is given in Listing 1. The vertex shader projects the vertices of a unit cube to clip space by multiplying them with the modelview and projection matrices. It also calculates the 3D texture coordinates for sampling and stores the object space positions for shading calculation.

The fragment shader gets the interpolated 3D texture coordinates and object space vertex positions. The current camera position (which is the eye ray's origin) is passed in as a shader uniform variable (line 11-12 in Listing 1). Using this eye ray's origin and the cube's interpolated object space position, the ray directions are obtained (line 14). Before running the costly sampling loop, we introduce an optimization which checks the current ray against the bounding box of the unit cube (line 16). Only if the ray intersects the bounding box, the sampling loop is initiated (lines 19-28); otherwise, the current fragment is discarded (line 30). This gives significant performance boost especially on the mobile platforms.

```
1  struct BBox {
2     vec3 min, max;
3  };
4  BBox getBBox(vec3 mn, vec3 mx) {
5     BBox temp;
6     temp.min = mn;
7     temp.max = mx;
8     return temp;
9  }
10 BBox bbox = getBBox(vec3(-0.5),vec3(0.5));
11 uniform vec3 eye_pos; //camera position
12 varying vec3 pos; //interpolated from vertices

13 void single_pass_raycaster() {
14    ray_dir    = normalize(pos - eye_pos);
15    frag_color = vec4(0.0);

16    if(intersects(eye_pos,ray_dir,bbox)) {
17       dir_step = ray_dir*steps;
18       ray_pos  = pos;
19       for(i=0;i<steps;i++) {
            //sampling and classification
20          sample = getSample(volume, ray_pos);
21          value  = texture1D(lut, sample);
            //compositing
22          pre_alpha=value.a-(value.a*
                              frag_color.a);
23          frag_color.rgb +=pre_alpha*value.rgb;
24          frag_color.a += pre_alpha;
25          ray_pos += dir_step; //advance ray
26          if(vec3(0)>ray_pos||ray_pos>vec3(1))
27             break;
28       }
29    } else {
30       discard;
31    }
32    return frag_color;
```

```
33}
```

Listing 1. Shader for the single-pass GPU ray casting

Depending on the current step size used and the total number of sampling points, a loop is initiated (line 19). In each iteration, the sample value is obtained by a lookup from the volume dataset (line 20). To the interpolated the value, the transfer function is applied (line 21) and then the classified value is accumulated using the current compositing scheme (line 22-24). The ray steps forward in the current viewing direction (line 25). The whole process is repeated until the ray exits the whole volume dataset (line 26-27). Finally, the composited color is returned as the fragment color (line 32).

With the versatility of the single-pass ray caster, we can modify the ray function instantly. Effects like maximum intensity projection (MIP), maximum intensity difference accumulation (MIDA), average, composite, and isosurface rendering modes can be realized easily thanks to the efficiency of the single-pass ray caster.

3) Isosurface Rendering with Adaptive Refinement

The isosurface can be given using the following implicit function

$$S = \{x \mid \varphi(x) = k\}$$

where, k is a constant and φ is called the embedding. The locality of a point p with respect to the isosurface depends on the value of $\varphi(p)$. If the value is less than k, the point is below the isosurface. If the value is greater than k, the point is above the isosurface. The shading calculation of the isosurface requires the evaluation of the normal vector. This normal vector is obtained from the normalized gradient of φ which is calculated as follows

$$N(x) = \frac{\nabla\varphi(x)}{|\nabla\varphi(x)|}$$

For volume dataset, the gradient is estimated using finite difference approximations such as centered finite difference as follows

$$\nabla f \approx \begin{bmatrix} \frac{f(x+1,y,z) - f(x-1,y,z)}{2} \\ \frac{f(x,y+1,z) - f(x,y-1,z)}{2} \\ \frac{f(x,y,z+1) - f(x,y,z-1)}{2} \end{bmatrix}$$

In case of isosurface rendering, only lines (22-24) are modified. Now instead of compositing, the current and the next sample's classified value is checked to see if the values are close to the given isovalue. If they are within an epsilon range of the isovalue, the normal is estimated using the centered finite difference approximation and then the shading is calculated using Blinn Phong shading model.

4) Combining Composite and Isosurface Rendering

We further developed a new hybrid rendering mode. This mode allows us to combine the result of multiple ray functions. For instance, we can combine the isosurface rendering to the composite rendering. The conventional isosurface is evaluated using a condition to test whether the current sample value is closer to the required isovalue. This results in an image containing a lot of sampling artifacts especially in regions where the isosurface blends with the background color, as shown by a rendering experiment in Fig. 3 (a). Moreover, since the isosurface is combined with the background color, the colors bleed on the isosurface, making its identification difficult.

In our solution, we eliminate the conditional test. Instead, we use blended isosurfaces which are generated using an alpha transfer function. This allows the isosurfaces to be blended naturally with the existing ray function. This avoids flickering artifacts in isosurface evaluation, because when the next ray step composites the current sample value, the isosurface loses its color. In addition, the resulting isosurface will not suffer from color bleeding artifacts, and therefore the isosurface color is sustained, as can been verified by the rendering experiment in Fig. 3 (b).

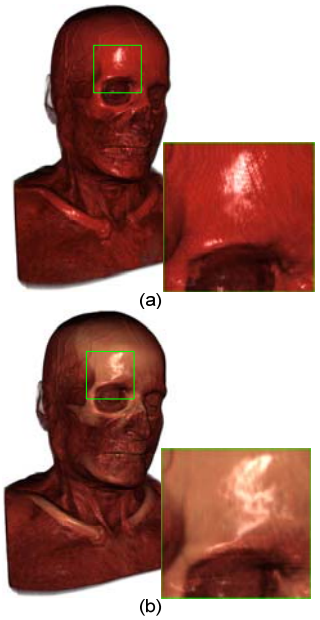

Figure 3. Ray casting for the Manix dataset: (a) without ray function blending, and (b) with ray function blending. (Note that both renderings used the same transfer function, but the color bleeding and sampling artifacts are removed in our ray function blended single-pass ray casting.)

5) Real-time Transfer Function Modification

For effective visualization, we provide a transfer function widget that can adjust the current transfer function as required in real-time. This widget was written entirely in the client side javascript.

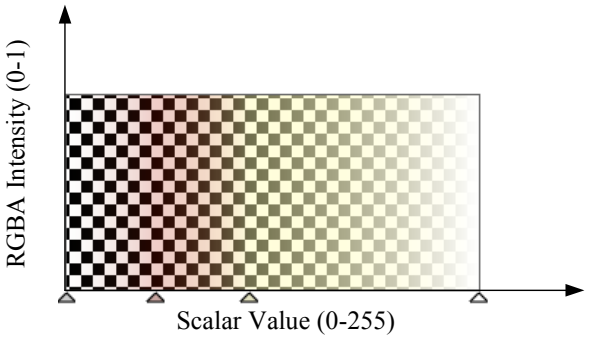

Figure 4. The transfer function widget for transfer function assignment in WebGL

The user can adjust the transfer function color and the current samples alpha value by assigning color and alpha values to specific keys. The X-axis corresponds to the scalar value (which for our case varies from 0 to 255 since we used 8-bit datasets) and the Y-Axis corresponds to the transparency value (which varies in the 0-1 range). Such a widget is shown in Fig. 4.

When the user modifies the transfer function by adding a new key or moving an existing key, first the new positions of all of the keys are determined. These keys are then sorted based on the intensity value they represent. Finally, the colors are interpolated and then the transfer function texture is updated. For efficiently modifying the transfer function texture, the gl.texSubImage2D function is used which directly modifies the texture data without creating a new texture. The whole process of transfer function modification is detailed in Listing 2. Each transfer function key contains a data-value pair with the data being intensity and the value being a color value containing red, green, blue and alpha components.

```
proc updateTF()
   //data contains the output transfer func. data
   sort the keys on the intensity
   for i=0 to keys.length
      dColor = keys[i+1].rgba-keys[i].rgba;
      dIndex = keys[i+1].intensity -
               keys[i].intensity;
      delta  = dColor/dIndex;
      for j=keys[i].intensity+1 to
            keys[i+1].intensity
         data[j] = data[j] + delta;
      end for
   end for
end proc
```

Listing 2. The transfer function modification pseudocode

IV. Experimental Results and Performance Assessment

The performance of the proposed algorithm was evaluated using a few experiments on two desktop platforms and two mobile platforms. Two different desktop systems were used which include the Dell Precision T7500 workstation (referred to as SYSTEM1) with a 2.27 GHz Intel Xeon CPU with 4 GB of RAM. This machine is equipped with an NVIDIA Quadro FX 5800 GPU with 4096 MB of dedicated video memory.

The second platform is a Dell Alienware M15x laptop (referred to as SYSTEM2) with an Intel Core i7 CPU and an NVIDIA GeForce 260M GPU. In addition, we also carried out experiments on two mobile platforms, an ACER Iconia A500 Tablet (referred to as MOBILE1), 1GHz dual-core Cortex A9 processor with an NVIDIA Tegra 2 GPU and a Samsung Galaxy SII GT-I9100 (referred to as MOBILE2), dual-core 1.2 GHz Cortex-A9 mobile phone with Mali-400MP GPU. Both of these mobile platforms ran the Google Android operating system.

For thorough evaluation, four 8-bit datasets were used in these experiments, namely Aorta (256×256×97), Skull (256×256×256), CTHead (256×256×256) and Manix dataset (256×230×256). These datasets were stored on our local web server. The datasets were stored into a 2D flat texture layout with an image resolution of 4096×4096. The Aorta dataset contained 96 slices which were stored in a 2D layout of 10×10. The other datasets contained 256 slices which were stored in a 2D layout of 16×16.

A. Comparison of Single-Pass Ray Caster to Multi-Pass Ray Caster on Desktop Platforms

The first set of experiments was conducted on both SYSTEM1 and SYSTEM2 to compare the performance of our single pass GPU ray caster against the current state-of-the-art WebGL ray caster [9]. For this experiment, we used the composite rendering mode with transfer function (shown in Fig. 4). All of the rendering and startup settings (for example the distance of the volume from the camera) were same for both of the WebGL ray casters. The tests were carried out on the canvas resolutions of 1024×1024 pixels. The ray sampling steps for this experiment were 100. These results are given in Table 1. The rendering results on the desktop platform in the Google Chrome Web browser are given in Fig. 5.

As can be seen from the statistics in Table 1, our proposed single pass ray caster consistently outperforms the multi-pass ray caster. The reason for this speedup in our proposed algorithm is the significantly less number of texture fetches and more efficient ray traversal. The multi-pass approach of [9] suffers from visual artifacts at the cube edges. Since we do not require any additional lookup (for example the multi-pass approach requires the front and back textures for extracting the ray direction), our performance remains consistent.

B. Comparison of Single-Pass Ray Caster to Multi-Pass Ray Caster on Mobile Platforms

The second set of experiments was carried out on the mobile platforms, for estimating the performance of the proposed GPU based single pass ray casting algorithm in WebGL on the mobile platforms. For this experiment, composite function without transfer function was used with 100 sampling steps. The reason we used 100 sampling steps is because we found out that the maximum steps for a variable loop supported on the tablet were 100. Any loop size larger than this value timed out the shader compiler and the tablet could not run the shader.

The canvas resolution for this experiment was 1024×1024. The datasets were down sampled to 1024×1024 resolution. This was due to the memory limitations on the mobile platforms. The results are given in Table 2 and the screenshots from the mobile platforms are shown in Fig. 6.

As can be seen, the performance trend we observed on the desktop platforms (shown in Table 1) is followed in this case as well. Our single pass ray caster out performs the multi-pass ray caster by almost 2x. We get much better performance even though all of the rendering settings are same. The reason for better performance in the single-pass ray caster seems to be due to the significantly less number of texture fetches as compared to the multi-pass approach. We expect the upcoming hardware to relax the variable length loop limit further which would improve the performance of this algorithm considerably.

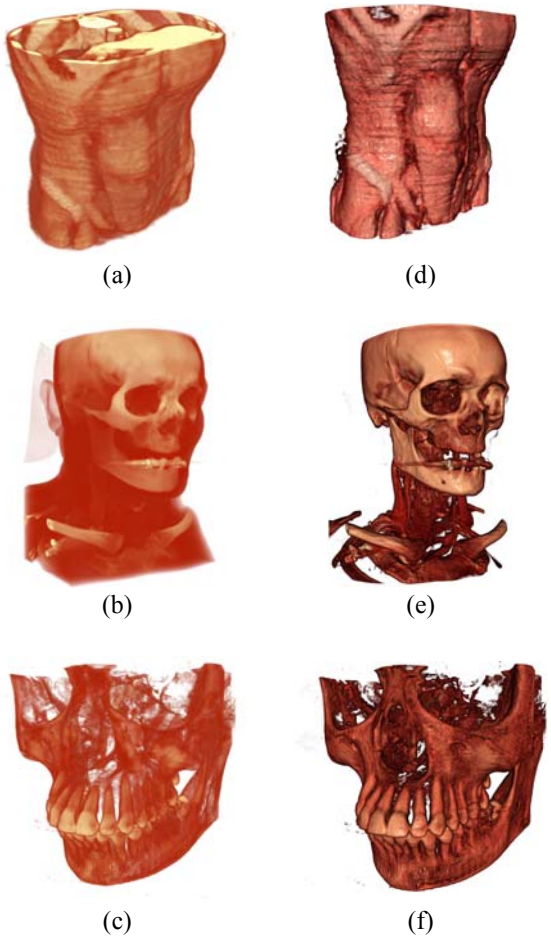

Figure 5. Rendering results from the proposed GPU-based single-pass ray casting on WebGL showing (a,d) the Aorta dataset, (b,e) the CTHead dataset, and (c,f) the Skull dataset. Figures (a-c) are generated using single pass ray caster with 256 sampling steps and composite rendering mode whereas Figures (d-f) are generated using 3D texture slicer with 512 sampling points and composite with shading rendering mode

Dataset	Hardware	Frame rate (in frames per second)	
		Multi-pass ray caster [9]	*Our proposed single-pass ray caster*
Aorta	SYSTEM1	72.3-90.7	90.8-90.9
	SYSTEM2	32.0-46.4	77.8-80.8
CTHead	SYSTEM1	52.3-80.7	89.4-90.9
	SYSTEM2	31.6-45.0	73.1-77.5
Skull	SYSTEM1	54.5-81.2	90.7-90.8
	SYSTEM2	31.0-44.7	75.6-77.8

Table 1. Comparison of performance of our proposed single pass GPU ray caster against the current state-of-the-art GPU ray caster for WebGL for 100 sampling steps on the desktop platforms.

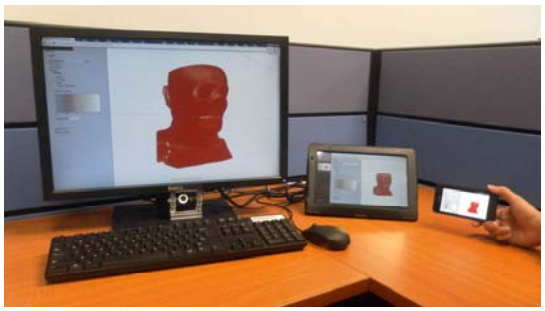

Figure 6. WebGL compliant volume rendering of 3D medical dataset implemented on the stand-alone desktop (left) and two mobile platforms: ACER Iconia A500 tablet (middle) and the Samsung Galaxy SII GT-I9100 mobile phone (right)

Dataset	Hardware	Frame rate (in frames per second)			
		Multi-pass ray caster [9]		*Our proposed single-pass ray caster*	
		512×512	1024×1024	512×512	1024×1024
Aorta	MOBILE1	0.5	0.1	0.9-1.0	0.2-0.3
	MOBILE2	1.5-1.8	0.5	4.4-4.6	1.3-1.6
CTHead	MOBILE1	0.5	0.1-0.2	1.0	0.2-0.3
	MOBILE2	1.8-1.9	0.5	3.3-3.6	1.0-1.1
Skull	MOBILE1	0.5	0.1-0.2	1.0	0.2-0.3
	MOBILE2	1.4-1.5	0.4-0.5	3.3-3.5	0.9-1.0

Table 2. Comparison of performance of our proposed single pass GPU ray caster against the current state-of-the-art GPU ray caster for WebGL for a sampling rate of 0.01 (100 sampling points) on the mobile platforms

C. Performance of 3D Texture Slicer on Desktop Platforms

In the third set of experiments, we evaluated the performance of 3D texture slicing on the desktop platforms. There were two rendering modes used in this experiment, the composite (as shown in Fig. 5 (a-c)) and the composite with shading mode (as shown in Fig. 5 (d-f)) which calculated the normal for shading by on demand gradient estimation using

centered finite difference. For all of the experiments on these platforms, a common canvas resolution of 1024 × 1024 pixels was used. The datasets were kept at 2 world units from the camera so that they were visible entirely on the screen. In addition to remove any biases due to a specific viewing direction, the datasets were rotated 720 degrees. The frame rates varied in a small range. There were three web browsers used in this experiment, Opera Next v 12.00 alpha, Google Chrome v 17.0.963.56 and Mozilla Firefox v 11.0 beta.

The Opera WebGL implementation uses the OpenGL API whereas both Chrome and Firefox WebGL implementations emulate the OpenGL ES API using DirectX 9 though the Angle engine. These WebGL implementations are named ChromeDX and FirefoxDX respectively. To disable the Angle rendering both Chrome and Firefox provide configuration settings. For Chrome, the –use-gl=desktop startup switch provides this whereas for Firefox, the configuration is manually set by setting the prefer-native-gl and force-enabled flags to true in the about:config configuration settings.

We named these configurations as ChromeGL and FirefoxGL in the experiments. The results for 256 sampling steps are presented in Table 3. We can see that there are significant differences in performance on different browsers. In our experiments, Google Chrome and Mozilla Firefox performed the best for all of the experiments.

Dataset	Web browser	Frame rate (in frames per second)	
		Composite	*Composite+Shading*
Aorta	Opera	80.1-90.0	66.9-74.6
	ChromeGL	96.3-97.5	60.6-70.9
	ChromeDX	95.8-97.7	60.8-70.9
	FirefoxGL	54.8-59.8	38.3-43.3
	FirefoxDX	94.9-100.0	65.2-75.2
CTHead	Opera	74.0-76.1	24.9-26.8
	ChromeGL	68.7-71.1	26.2-30.7
	ChromeDX	68.9-71.4	26.7-30.1
	FirefoxGL	41.7-43.5	21.6-24.0
	FirefoxDX	76.9-82.3	28.5-31.2
Skull	Opera	73.5-80.1	24.2-26.9
	ChromeGL	71.6-74.4	26.7-31.4
	ChromeDX	70.9-74.8	26.9-31.7
	FirefoxGL	32.4-43.4	21.1-24.8
	FirefoxDX	80.8-85.5	28.3-33.1

Table 3. The performance results for 3d texture slicing in WebGL for a sampling rate of 0.0039 (256 sample points) on SYSTEM1

D. Comparison of Single-pass Ray Caster to 3D Texture Slicer on Desktop Platforms

The fourth set of experiments was conducted on both SYSTEM1 and SYSTEM2 to compare the performance of the single pass GPU ray caster against the 3D texture slicer in WebGL on the Google Chrome browser. For this experiment, the composite rendering mode with transfer function was used. The tests were carried out on the canvas resolutions of 1024×1024 pixels.

The ray sampling steps as well as the total texture slices were both 256 and the volume was placed 2 units from the camera so that it covered the whole screen. These results are given in Table 4. As can be seen from the statistics in Table 4, the performance of 3D texture slicer is almost 1.5x better as compared to the single pass ray caster.

E. Comparison of Single-pass Ray Caster to 3D Texture Slicer on Mobile Platforms

In the final set of experiments, we compared the performance of 3D texture slicer and single pass GPU ray caster on the mobile platforms. The canvas resolution was 1024×1024. For this experiment, composite function without transfer function was used. The total sampling steps used were 100 for both texture slicing and GPU ray casting because the fragment shader could not link on the tablet for variable loop size larger than 100. These results are given in Table 5.

As can be seen, we were able to obtain interactive frame rates even on a mobile platform. The 3D texture slicer outperformed the GPU ray caster by almost 2x. The results concluded that performance wise, for volume rendering, 3D texture slicing is consistently better both on the desktop as well as the mobile platforms.

Dataset	Hardware	Frame rate (in frames per second)	
		Ray caster	*3D texture slicer*
Aorta	SYSTEM1	63.6-79.4	98.2-98.7
	SYSTEM2	33.5-35.2	51.3-55.2
CTHead	SYSTEM1	39.3-47.5	54.9-59.4
	SYSTEM2	16.9-18.4	29.2-32.5
Skull	SYSTEM1	33.7-41.0	58.4-61.8
	SYSTEM2	14.2-19.5	30.6-35.5

Table 4. Comparison of performance of the single pass GPU ray caster against the proposed 3D texture slicer for WebGL on the desktop platforms

Dataset	Hardware	Frame rate (in frames per second)	
		Ray caster	*3D texture slicer*
Aorta	MOBILE1	1.2-1.3	2.3-2.5
	MOBILE2	1.3-1.6	3.8-5.6
CTHead	MOBILE1	0.2-0.3	1.9-2.4
	MOBILE2	1.0-1.1	4.3-5.1
Skull	MOBILE1	0.2-0.3	2.4-2.5
	MOBILE2	0.9-1.0	4.8-5.1

Table 5. Comparison of performance of 3D texture slicer and GPU ray casting for a sampling rate of 0.01 (100 sample points) on mobile platforms

V. DISCUSSION AND CONCLUSION

We have presented the first single-pass ray caster for WebGL. In contrast to the current state-of-the-art for volume rendering in WebGL [9], our implementation is able to handle dynamic transfer functions even on mobile platforms. A quantitative evaluation revealed that the proposed algorithm outperforms the existing WebGL ray caster by up to 2x on both stand-alone and mobile platforms. Such high

performance is helpful especially on the mobile platforms where the texture accesses are too costly. With the new mobile devices, we expect more support in the upcoming WebGL implementations for more advanced shaders.

Currently, due to the limitation in the loop iteration for dynamic loops on the mobile platforms, it is not possible to implement advanced shaders. Therefore, to have high performance volume rendering especially on the mobile platform, we presented the first 3D texture slicer for WebGL. This allows us to render volumes with higher sampling rates on the mobile platforms. The experimental results have shown that, performance wise, 3D texture slicer performs better as compared to the single-pass ray caster.

While both of them are suitable for WebGL implementation, the choice is subject to a performance/quality trade off. Although, the limited support for variable loops in the current WebGL implementations on the mobile platforms prevents us from introducing more optimization in the single-pass ray caster, we expect that the upcoming hardware will relax such restrictions. This will enable the proposed single-pass ray caster to perform better on the upcoming mobile platforms.

We are confident on the results obtained from the experiments and would like to expand the work to address specific applications such as rapid prototyping of biomedical models [10], coupling between deformation and rendering of volumetric models [11,12,19] as well as confocal endomicroscopy [15]. This will enable the ubiquitous visualization and processing capabilities in a wide application domain.

In conclusion, thanks to the wide availability of the WebGL architecture, we have successfully developed a ubiquitous volume renderer for visualization of the biomedical datasets directly on the mobile platforms. With new and improved hardware features in the upcoming mobile devices, we expect these advanced features to be exposed through WebGL. This would allow a richer interactive visualization experience. Moreover, with new and improved mobile GPUs in the coming generations, we expect our algorithm to perform even better on these newer devices. WebGL is indeed a promising platform for high-quality mobile applications. We can carry out more sophisticated renderings, for example, dynamic shading using real-time texture lookups in 3D texture slicing even on the mobile platforms. Such effects are impossible in the current state-of-the-art ray caster.

ACKNOWLEDGMENT

This work is partially supported by two research grants, M408020000 from Nanyang Technological University and M4080634.B40 from Institute for Media Innovation, NTU, and a grant MOE2011-T2-2-037 from Ministry of Education, Singapore.

REFERENCES

[1] C. Marrin, “The official WebGL specifications,” Available online: http://www.khronos.org/registry/webgl/specs/latest/, accessed in 2012.

[2] K. Engel, M. Hadwiger, J. M. Kniss, A. Lefohn, C. R. Salama and D. Weiskopf, “Real-time volume graphics,” A.K.Peters Publisher, 2005.

[3] B. Preim and D. Bartz, "Visualization in Medicine," Elsevier Inc. Publisher, 2007.

[4] M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski, “Advanced illumination techniques for gpu-based volume raycasting,” ACM SIGGRAPH 2009 Courses, 2009.

[5] J. Kruger and R. Westermann, “Acceleration techniques for GPU-based volume rendering,” Proceedings of the 14th IEEE visualization, 2003.

[6] S. Stegmaier, M. Strengert, T. Klein and T. Ertl, “A simple and flexible volume rendering framework for graphics-hardware-based raycasting,” Proceedings of The Fourth International Workshop on Volume Graphics, 2005, pp. 187 – 241.

[7] S. Bruckner and M. E. Groller, “Instant volume visualization using maximum intensity difference accumulation,” Computer Graphics Forum, 28(3), 2009, pp. 775-782.

[8] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler and M. Gross, “Real-time ray-casting and advanced shading of discrete isosurfaces,” Computer Graphics Forum, 24(3), 2005, pp. 303-312.

[9] J. Congote, L. Kabongo and A. Moreno, “Interactive visualization of volumetric data with WebGL in real-time,” Proceedings of The 2011 Web3D ACM Conference, 2011, pp. 137-146.

[10] F. Lin, H. S. Seah, Z. Wu, D. Ma, “Voxelisation and fabrication of freeform models,” Virtual and Physical Prototyping, vol. 2(2), 2007, pp. 65-73.

[11] M. M. Movania and F. Lin, “A Novel GPU-based Deformation Pipeline,” ISRN Computer Graphics, Vol. 2012, Article ID: 936315, 2012. doi:10.5402/2012/936315.

[12] M. M. Movania and F. Lin, “Real-time Physically-based Deformation on the GPU using Transform Feedback,” Chapter 17 in The OpenGL Insights, AK Peters/CRC Press, pp:233-248, 2012.

[13] M. M. Movania and F. Lin, “Ubiquitous Medical Volume Rendering on Mobile Devices” IEEE International Conference on Information Society (i-Society 2012), London, UK, pp:93-98, 2012.

[14] M. M. Movania and F. Lin, “High-Performance Volume Rendering on the Ubiquitous WebGL Platform” The 14th International Conference on High Performance Computing and Communication (HPCC 2012), Liverpool, UK, June 2012 (in press).

[15] P. Thong, M. Olivo, S. Tandjung, M. M. Movania, F. Lin, K. Qian, H. S. Seah, K. C. Soo, “Review of Confocal Fluorescence Endomicroscopy for Cancer Detection,” IEEE Photonics Society (IPS) Journal of Selected Topics in Quantum Electronics, PP(99), 2011, pp:1-12.

[16] T. J. Cullip and U. Neumann, "Accelerating volume reconstruction with 3D texture hardware," University of North Carolina at Chapel Hill, Tech. Rep.,1994.

[17] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in Proc. Symp. Volume Vis., Tysons Corner, Virginia, United States, 1994, pp. 91-98.

[18] A. Van Gelder and K. Kim, "Direct volume rendering with shading via three-dimensional textures," in Proc. Symp. Volume Vis., San Francisco, California, United States, 1996, pp. 23-30.

[19] M. M. Movania, F. Lin, K. Qian, W.M. Chiew and H.-S. Seah, "Coupling between Meshless FEM Modeling and Rendering on GPU for Real-time Physically-based Volumetric Deformation", Journal of WSCG, Vol. 20, No. 1, pp:1-10, ISSN 1213-6972, Union Agency, 2012.