Visualization of Very Large 3D Volumes on Mobile Devices and WebGL

José M. Noguera, Juan-Roberto Jiménez
Graphics and Geomatics Group of Jaén. University of Jaén.
Campus Las Lagunillas, Edificio A3, 23071 Jaén, Spain.
{jnoguera,rjimenez}@ujaen.es

ABSTRACT

Platforms based on OpenGL ES 2.0 such as mobile devices and WebGL have recently being used to render 3D volumetric models. However, the texture storage limitations of these platforms cause that only low-resolution models can be visualized. This paper describes a novel technique that overcomes these limitations and allows us to render detailed high resolution volumes on these platforms. Additionally, we propose a software architecture that permits existing volume rendering techniques to be adapted to mobile devices and WebGL. A set of experiments has been carried out to assess the performance of the proposed architecture on these platforms with different volumes of increasing resolution. Results prove that our proposal is feasible, robust and achieves visualization of very large volumes on constrained platforms.

Keywords: Volume visualization, OpenGL ES, mobile devices, WebGL, large volumetric models, software architecture.

1 INTRODUCTION

Nowadays mobile devices are extensively used as a worthy tool in many different scenarios of our life. Their hardware and software capabilities are constantly being enhanced, and recent research has demonstrated their validity to compute complex computer graphics algorithms. In fact, it has been proved that the volume visualization field can benefit from the properties of mobile devices in many interesting applications [18, 1, 5, 7, 9, 11, 13, 22].

However, it is a common misunderstanding to assume that the same results can be achieved by a literal translation to mobile devices of classic algorithms originally developed for standard PCs or workstations. There are two important factors that must be taken into account:

- The standard graphics specification of this kind of devices is OpenGL ES 2.0 [12], which differs from the desktop PC counterpart in several aspects, e.g., the lack of 3D texture support.
- These devices must rely on batteries, so their hardware and software architectures are designed to favour power-efficiency instead of pure computing power.

In addition, recent advances in display technologies allow today's mobile devices to feature large and high resolution screens, which require large volumetric models in order to achieve a minimum of quality. For example, newer tablets such as the iPad3 feature screen resolutions that surpass the Full-HD standard used in most monitors and TV screens. Nevertheless, their GPU and memory capacities are still limited and do not support large models directly.

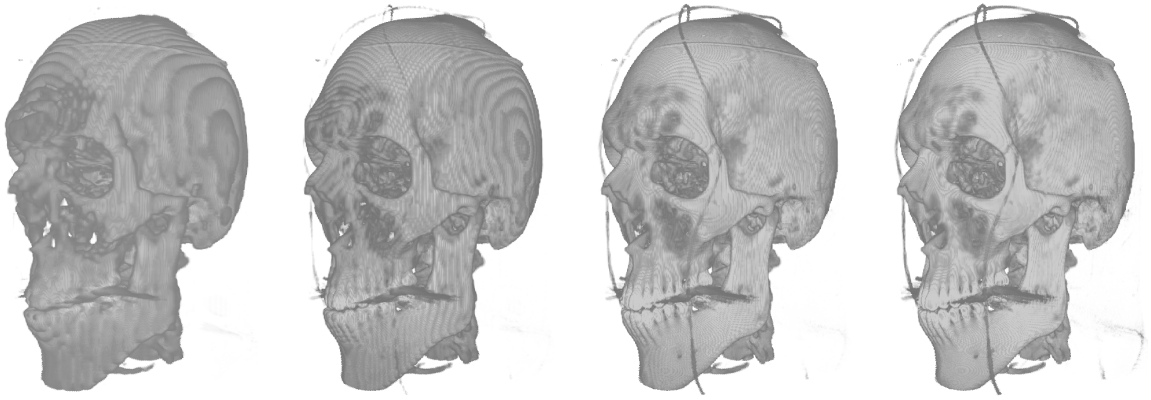
In this paper, we deeply study this problem in the context of direct volume rendering. We present a proposal to render very large volumetric models in order to meet the user expectations in quality and performance by overcoming the referred limitations of handheld devices, see Figure 1. Moreover, we describe a software architecture that allows us to adapt existing volume rendering techniques based on 3D textures to platforms that only support 2D textures.

The ideas described in this paper are also applied to WebGL[1], the standard for accelerated graphics on the Web. As this standard is based on the same specification used by mobile devices, i.e. OpenGL ES 2.0, it suffers from the same limitations, including the lack of 3D textures.

Finally, we have implemented a mobile and a WebGL based prototypes and conducted a set of experiments to test performance of these platforms under the conditions of maximum storage requirements.

The rest of the paper is organized as follows. Section 2 presents current research in volume visualization techniques for mobile devices and WebGL. In Section 3,

[1] https://www.khronos.org/registry/webgl/



64×64×256 128×128×256 256×256×256 512×512×256

Figure 1: Volumes of increasing resolution. Images rendered using the proposed technique.

our novel technique and architecture for volume rendering are presented. Section 4 presents our performance evaluation and discusses the results. Finally, Section 5 concludes the paper.

2 PREVIOUS WORK

In the context of scientific visualization and volume rendering, a volume is usually represented as a set of images/slices that are parallel and evenly distributed across the volume. Equation 1 expresses the computation of the final color of a given pixel by composing the colors and opacities of the samples along a given line across the volume, for a certain wavelength λ [10]:

$$C_\lambda(x) = \sum_{k=0}^{n} c_\lambda(x+r_k)\alpha(x+r_k) \prod_{l=k+1}^{n} (1-\alpha(x+r_l)) \quad (1)$$

where $C_\lambda(x)$ is the final color at a given position x, $c(x+r_k)$ is the color of the kth sample at position $x+r_k$ inside the volume and $a(x+r_k)$ is its corresponding opacity.

Volume visualization algorithms have not been applied to mobile devices until recently. First attempts overcame the mobile devices limitations by employing a server-based rendering approach. This approach relies on a dedicated rendering server that carries out the rendering of the volume and streams the resulting images to the mobile client over a network [7, 9]. Also following a server-client scheme, Zhou et al. [22] employed a remote server to precompute a compressed iso-surface, which is sent to the mobile device allowing a faster rendering. Moser and Weiskopf [11] introduced an interactive technique for volume rendering on mobile devices that adopts the 2D texture slicing approach. Noguera et al. [13] proposed an algorithm that overcomes the 3D texture limitation of mobile devices and achieves interactive frame rates by caching the geometry of the slices in a vertex buffer object (VBO). ImageVis3D [5] is an iOS application that uses the 2D texture slicing approach. While the user is interacting the number of slices is drastically reduced. At the end of an interaction a new image is rendered with the whole set of slices. This rendering step is carried out in the mobile device itself, or in a remote server in case of complex or large models. Focused on the visualization of bones, Campoalegre [18] also proposed a client-server scheme where the model is compressed in the server side by the Haar Wavelet function and reconstructed in the client device. On the other hand, Congote et al. [1] implemented a ray-based technique using the WebGL standard.

All the aforementioned techniques share the same limitation: the lack of 3D textures on OpenGL ES 2.0 and WebGL severely restricts the size and resolution of the volumetric models that can be rendered. The problem is aggravated on WebGL, as these applications are usually run on desktop computers equipped with large monitors.

The following proposals deal with the problem of very large volumetric models but in the context of PC or workstations whose features differ from the intrinsic peculiarities of mobile devices and WebGL. A straightforward method to deal with a large volume is the bricking technique [2]. This technique subdivides the volume into several smaller blocks in such a way that a single block fits into texture memory. Gunthe and Straßer [3] used a wavelet based volume compression in order to render large volume data at interactive frame rates in a standard PC. Tomandl et al. [17] combined local and remote 3D visualization (standard PC + high-end graphics workstation) achieving low-cost but high-quality 3D visualization of volumetric data. Schneider and Westermann [15] also overcame the problem of the limited texture memory by compressing large scale volumetric data sets. Their solution takes advantage of temporal coherence on animated environments. Thelen et al. [16] introduced a dynamic subdivision scheme incorporating multi-resolution wavelet representation to

visualize data sets with several gigabytes of voxel data interactively on distributed rendering clusters. Finally, Xie et al. [21] subdivided the volume dataset into a set of uniform sized blocks and combined early ray termination, empty-space skipping and visibility culling techniques to accelerate the rendering process.

3 METHODOLOGY

This section details the algorithm, the software architecture and the implementation details that we propose to render large volumetric models on handheld devices and WebGL. Volumes are usually stored as a set of slices, each one containing a 2D image that represents the intersection of the volume with the slice. Common volume rendering approaches [19] store these slices in a 3D texture. However, neither mobile devices nor WebGL support 3D textures. This limitation can be overcome by storing the slices in a single 2D texture following a mosaic configuration [1, 13]. Nonetheless, without recurring to external servers, this technique limits the size of the volume that can be stored because 2D textures are considerably smaller than 3D textures.

We extend this mosaic configuration solution to exploit the maximum texture capacity of the GPU in order to deal with larger volumetric models. Our idea is based on maximizing the multi-texture storage capacity of the device by using all the available texture units and color channels. Usually, current handheld devices are able to store up to 8 RGBA textures of 2048^2 texels each. These numbers give us a maximum volume size of 512^3 voxels when using our technique, which is considerably larger than the models rendered until now on mobile platforms.

Our technique stores the 3D volume by placing each slice one next to the other in a given color channel of a 2D texture. If the texture dimensions are exceeded, we continue storing slices in the next color channel. This way, data-level parallelism is optimized [6, 20]. When all the channels of the texture are completed, the remaining slices are stored in consecutive texture units following the same pattern. Figure 2 shows an example of a 2D texture where each color channel stores a subset of slices in a mosaic configuration. Thus, each RGBA color represents the values inside four different non-consecutive slices of the volume.

Our storage configuration technique can be utilized with any standard volume rendering approach based on 3D textures. Figure 3 illustrates our proposal for a volume rendering architecture designed for OpenGL ES 2.0. This architecture is divided into two main parts: the *texture memory* and the *shader*. The texture memory is used to store both the 3D volume and the transfer function. The volume is stored using multi-textures as previously described. On the other hand, the transfer function refers to the texture normally required

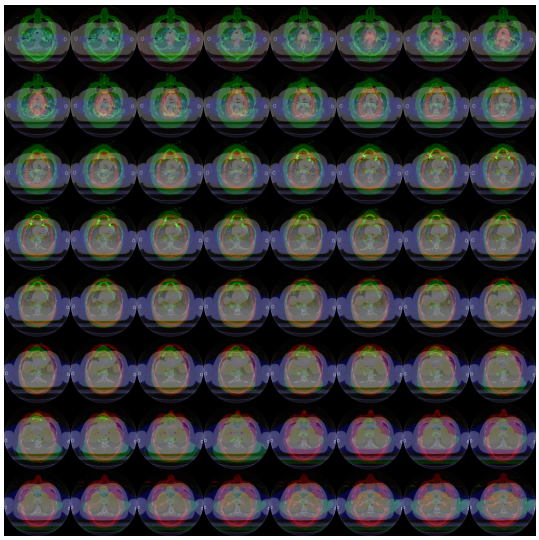


Figure 2: Four mosaics stored in an RGBA texture.

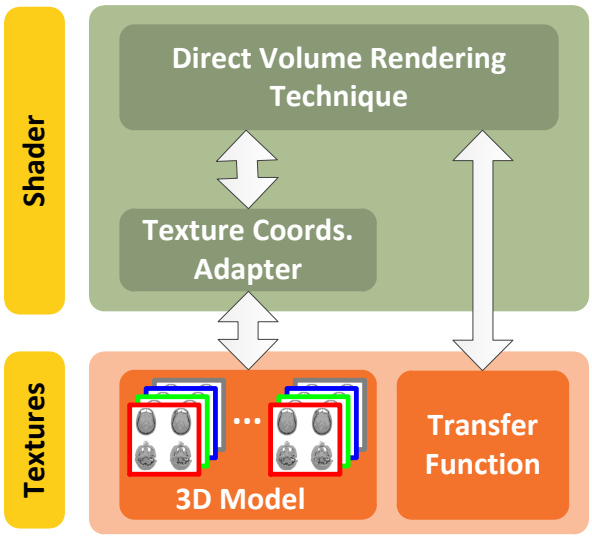


Figure 3: Software architecture of our proposal.

by volume rendering techniques to assign a color to each voxel [19]. Thereby, different parts of the model (bones, muscles, etc.) can selectively be emphasized by interactively modifying the transfer function.

In the shader section there are two modules: the *texture coordinate adapter* (TCA) and the selected *direct volume rendering* (DVR) technique. The DVR technique sends a 3D coordinate to the TCA and receives back an interpolated grey-tone. This tone is then converted into an RGBA value according to the given transfer function and used to compute the color of the corresponding fragment. It is important to remark that the DVR technique is unaware of the underlying 3D model storage method. In fact, this architecture provides an straightforward mechanism to adapt existing DVR techniques to the platforms we are interested in this paper.

$$
\begin{aligned}
aux_1 &= floor(S * z) \\
aux_2 &= floor\left(\frac{aux_1}{M_x M_y}\right) \\
depth &= min\left(1.0, \frac{M_x M_y}{S}\right) \\
z_{ini} &= aux_2 * depth \\
z_{res} &= \frac{z - z_{ini}}{depth} \\
u_1 &= floor\left(\frac{aux_2}{Max_{ch}}\right) \\
ch_1 &= mod(aux_2, Max_{ch})
\end{aligned}
$$

Listing 1: Computation of the texture unit u_1, the color channel ch_1 and the corresponding (x, y, z_{res}) from the 3D texture coordinate (x, y, z).

The TCA module transforms the 3D texture coordinate provided by the DVR technique to a format suitable for the volume representation and computes the grey-tone. The returned value includes the trilinear interpolation with the one-voxel distance neighbourhood. The process performed by this module can be decomposed into two differentiate tasks.

The first task consists of deriving the texture unit u_1, the RGBA channel ch_1 and the new local 3D texture coordinates (x, y, z_{res}) from the original 3D texture coordinates (x, y, z). Listing 1 shows how to compute these values. S is the number of slices in the volume, $M_x \times M_y$ is the maximum number of slices in a mosaic stored in a single color channel of a texture, see Figure 2, and Max_{ch} is the number of channels per texture. As the slices are stored in a consecutive manner along the channels and texture units, each mosaic stores an interval of the z-component of the full volume. The value z_{res} is the residual z defined as the original z minus the z coordinate of the voxel stored in the first texel of the selected mosaic.

The second task is devoted to compute a pair of 4-tuples (u_1, ch_1, x_1, y_1), (u_2, ch_2, x_2, y_2) that define two texels from the set of 2D textures, where u_i, ch_i refers to the texture unit and the color channel, respectively, and x_i, y_i are the 2D texture coordinates of the desired texel in the corresponding mosaic stored in the texture unit u_i. These two texels are neighbours along the z direction, and are placed in the same or in consecutive mosaics. The grey-tones of these texels are merged to simulate trilinear interpolation. Note that bilinear interpolation is automatically obtained by the texture interpolator of the GPU. Listing 2 shows how to perform these operations. Here, T is an array of 2D samplers that contains the volumetric model.

$$
\begin{aligned}
S_{tex} &= min(M_x M_y, S) \\
aux_3 &= floor(z_{res} * S_{tex}) \\
aux_4 &= mod(aux_3 + 1, S_{tex}) \\
x_1 &= \frac{aux_3}{M_x} + \frac{x}{M_x} \\
y_1 &= \frac{floor\left(\frac{aux_3}{M_y}\right)}{M_y} + \frac{y}{M_y} \\
x_2 &= fract\left(\frac{aux_4}{M_x}\right) + \frac{x}{M_x} \\
y_2 &= \frac{floor\left(\frac{aux_4}{M_y}\right)}{M_y} + \frac{y}{M_y} \\
next &= c + step(aux_3, aux_4) \\
u_2 &= t + step(Max_{ch}, next) \\
ch_2 &= mod(next, Max_{ch}) \\
v_1 &= tex2D(\ T[u_1],\ (x_1,\ y_1)\ )[ch_1] \\
v_2 &= tex2D(\ T[u_2],\ (x_2,\ y_2)\ )[ch_2] \\
V &= mix(v_1,\ v_2,\ z_{res} * S_{tex} - aux_3)
\end{aligned}
$$

Listing 2: Computation of V, the value of the grey-tone at position (x, y, z).

The shaders represented by Listings 1 and 2 have carefully been designed in order to avoid flow control operators, when possible, by promoting the use of built-in GLSL functions like *step*. Observe that the use of flow control operators have a cost in the GPU of mobile devices.

4 RESULTS

In order to measure the effectiveness and performance of our technique two prototypes have been implemented. The first one is a mobile application using OpenGL ES 2.0 and the second one is a desktop solution using WebGL. The selected technique for the DVR module is a ray-based technique implemented in the GPU [4, 8]. This technique basically consists of a loop of n steps that traverses the volume accumulating color and opacities along a given ray-direction.

Recall that our architecture is independent of the visualization technique, and a faster texture-based approach could have been used instead [13]. However, our goal was not to measure the performance of the DVR technique, but to assess the performance and scalability when the resolution of the 3D model increases and multiple textures are used.

In our experiments, we used the *CT human* male dataset provided by the *Visible Human Project*[2]. This dataset has a total resolution of $512^2 \times 1877$ voxels.

4.1 Results on Mobile Devices

The experiments were conducted on an iPad2 tablet. This device features a dual core PowerVR SGX543MP2 GPU and the iOS 5 operating system. The test application was developed as a native iOS application, using C++ and GLSL ES 2.0. According to Apple's technical specifications, this device supports a maximum 2D texture size of 2048^2 texels, and up to 8 texture units.

Our experiments intended to cover all the range of model resolutions provided by our technique. We used a subset of the CT human dataset, shown in Figure 4, with different resolutions. For each experiment, a 100 frame animation of the camera rotating around the CT human model was generated, and the mean times needed to render each frame were taken. Due to the tile-based deferred rendering architecture [14] used by the GPU, OpenGL ES calls can be deferred until the scene is presented. In order to perform exact timings, we forced the frame rendering to finish by means of a *glFinish* call. Figure 5 shows graphically the results obtained in milliseconds. The results for the following experiments are included:

- 128^3A: stored using one single-channel texture.
- 128^3B: the same model as the previous experiment.
- 256^3A: stored using one RGBA texture.
- 256^3B: stored using four single-channel textures.
- $512^2 \times 384$: stored using six RGBA textures.

The experiment 128^3A utilized a simplified version of the TCA module that only handles one mosaic, similar to the proposal of Congote et al. [1], while the experiments 128^3B, 256^3A, 256^3B and $512^2 \times 384$ used our proposed TCA module that can deal with high resolution 3D models.

Note that $512^2 \times 384$ is the maximum resolution that can be achieved by our technique on this device, because one texture unit is used by the transfer function and another one is required by the rendering technique.

In all cases, uncompressed 2048^2 textures were used. The ray-based DVR technique performed 80 steps in all the experiments. The screen resolution was 480×320 pixels.

Results in Figure 5 show that the rendering times are almost constant among all the tested datasets when using the same TCA module. This suggests that the resolution of the volumetric model does not have a significative impact on the performance of the rendering process, as long as the model fits in the device's memory. The performance of the volumetric rendering depends on the screen resolution and on the number of steps performed during the raycasting rather than on the model resolution, as it was studied in [1, 13].

As stated above, the experiments 128^3A and 128^3B were performed using the same dataset but different TCA modules. The experiment 128^3B used our technique to compute the texture coordinates whereas 128^3A used a simplified one. It was possible to use this simplified version because the model is small enough to fit in one mosaic. As shown in Figure 5, the rendering time of the second experiment doubles the time achieved by the first one. Our proposed technique neither increases the number of texture accesses nor

Figure 4: The CT human model on an iOS mobile device. Resolution 256^3 using 4 textures and 80 steps-raytracing.

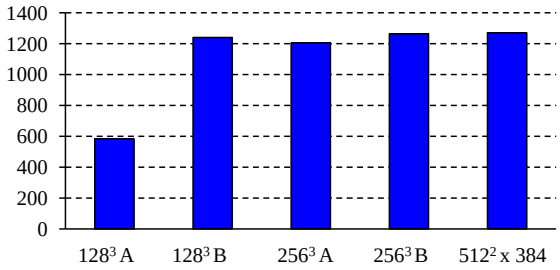


Figure 5: Rendering time (ms) for an iPad2 mobile device. Screen resolution: 320×480 pixels. Raytracing: 80 steps.

[2] http://www.nlm.nih.gov/research/visible/visible_human.html

adds additional conditional branches in the shader. Nevertheless, it increases its longitude by about a dozen of straightforward code lines to handle additional color channels and texture units, see Listings 1 and 2. Albeit these lines only perform simple computations, they are repeated once per step.

The experiments 256^3A and 256^3B employed a dataset that is too large for one mosaic of 2048^2 pixels. Two strategies can be used to store it: we can use either one RGBA texture (256^3A) or four single-channel textures (256^3B). Our experiments show that there are no significative differences between both strategies in terms of rendering times, and as a consequence, we can use the best suited for our particular application.

Finally, we also prove that our technique allows us to exploit all the available texture resources of the mobile device in order to render a very large dataset. We manage to render a model of up to $512^2 \times 384$ voxels while keeping the same rendering speed.

4.2 Results on WebGL

Following, we conducted a similar set of experiments to test the performance of the WebGL implementation. The tests were carried out using an Intel Core2 Quad CPU Q6600, 4 GB of RAM, a GeForce 8800GT and Windows 7 SP1 64 bits. As web browser, we tested Opera 11.50 labs (build 24661).

The selected GPU supports 2D texture sizes of up to 8192^2 texels and provides 16 texture units. Given that the texture size is considerably larger than the provided by the iPad2, we used a larger subset of the CT human dataset for our experiments, as shown in Figure 6.

In order to measure the rendering times, we forced the WebGL canvas to redrawn continuously and counted the number of frames rendered during an animation. This animation consisted of the camera rotating around the model during 5 seconds. Figure 7 shows graphically the mean times needed to render each frame in milliseconds. The results for the following experiments are included:

- $512^2 \times 256$A: stored using one single-channel texture.
- $512^2 \times 256$B: the same model as the previous experiment.
- $512^2 \times 1024$: stored using one RGBA texture.

The experiment $512^2 \times 256$A utilized the simplified version of the TCA module described in Section 4.1, while the experiments $512^2 \times 256$B and $512^2 \times 1024$ used our proposed TCA module. In this case, we used uncompressed 8192^2 textures and 128 steps for the ray-casting. The WebGL canvas resolution was 800^2 pixels.

We found that Opera stopped working every time we tried to load more than one 8192^2 texture. The reason was that the NVIDIA driver exceeded the Windows imposed rendering time limit (TDR) of two seconds. This limited our experimentation to a model of $512^2 \times 1024$ voxels, which is the maximum size that can be encoded using all color channels of a single 8192^2 RGBA texture.

Interestingly, we did not run into these problems when we were experimenting with mobile devices, in spite of the fact that both platforms share the OpenGL ES 2.0 specification. In our opinion, today's WebGL implementations are still relatively immature and the tested mobile device proved to be a more predictable and stable platform. As opposed to WebGL under Opera, the iPad2 was able to correctly handle all our experiments, including those using the maximum texture size on all the available texture units.

Nevertheless, comparing Figures 5 and 7 we can easily observe that WebGL is more than one order of magnitude faster than the selected mobile device when rendering a similar volumetric model, even with a commodity desktop PC.

The experiments $512^2 \times 256$A and $512^2 \times 256$B (see Figure 7) used a model that can be stored in a single mosaic. Therefore, our proposed TCA module was not strictly needed for such a small model. As stated above, both experiments differed in the TCA module. The difference in time shows the cost of including our module to deal with large models. We can clearly see that the GPU handles the additional operations without a noticeable increment of time.

The last experiment ($512^2 \times 1024$) used a large model that cannot be encoded on one mosaic in a conventional way. Therefore, our TCA module is mandatory to render it. In this case, the four color channels of a texture were used, and thus, the model was four times larger than the one used in the previous experiments. This experiment showed that the rendering time is greater than in the previous experiments. This result is somewhat a surprise, since the texture dimensions, operations and texture fetches are the same. In fact, the only difference is the number of color channels.

5 CONCLUSIONS

Due to the today's mobile GPU limitations, it was not possible to render volumetric models larger than 128^3 voxels on devices such as the Apple's smartphones and tablets. However, in this paper we have proposed a novel technique that enables mobile devices to render very large volumes by using multi-texturing to encode volumetric models on a set of RGBA 2D textures.

We have also proposed a simple and easy to implement architecture that can be used to adapt any existing di-

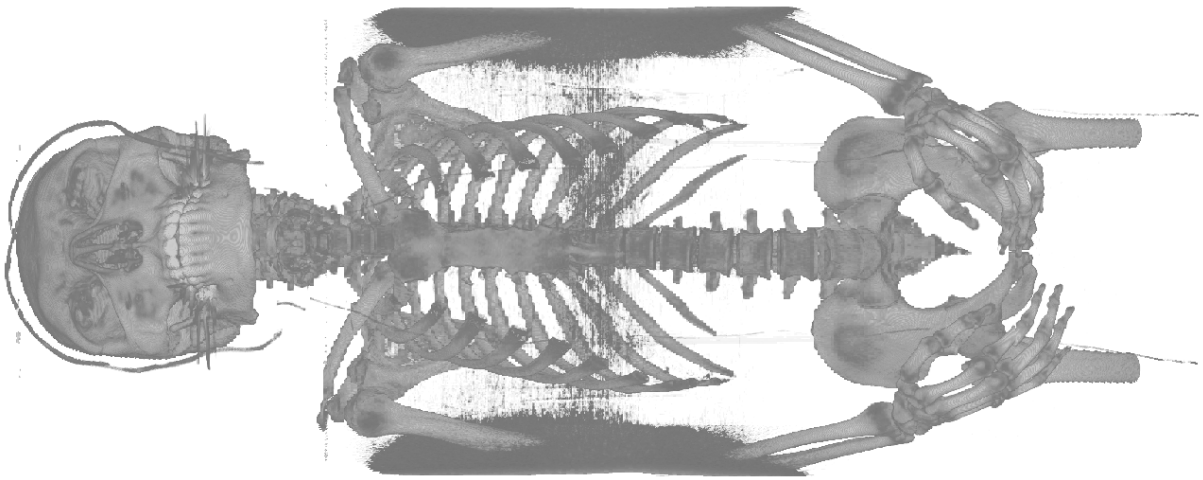


Figure 6: Visible human dataset rendered with WebGL, resolution: $512^2 \times 1024$ voxels.

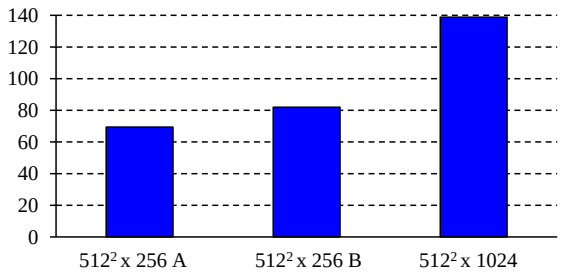


Figure 7: Rendering time (ms) for a desktop PC with a nVidia Geforce 8800GS. Screen resolution: 800^2 pixels. Raytracing: 128 steps.

rect volume rendering technique based on 3D textures to mobile devices and WebGL.

Our experiments have proved that we can render volumes of up to $512^3 \times 384$ voxels on a mobile device without decreasing the rendering speed. The proposed technique is also very akin to WebGL, because this standard shares the same limitations that mobile devices, mainly the lack of 3D texture support.

Regarding future works, we plan to study texture compression in order to reduce cache issues and improve efficiency. Furthermore, we plan to test the performance problems when transmitting large volumes across the Internet on WebGL. We want to explore multiresolution techniques in order to optimize network bandwidth. Progressive refinement techniques can probably be used in this context to improve the user interaction experience with the volume.

ACKNOWLEDGEMENTS

This work has been partially supported by the Spanish "Ministerio de Ciencia e Innovación" and the European Union (via ERDF funds) through the research project TIN2011-25259; by the "Consejería de Innovación, Ciencia y Empresa" of the "Junta de Andalucía" and the European Union (via ERDF funds) through the research project P07-TIC-02773; and by the University of Jaén through the project PID441012.

6 REFERENCES

[1] J. Congote, A. Segura, L. Kabongo, A. Moreno, J. Posada, and O. Ruiz. Interactive visualization of volumetric data with WebGL in real-time. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, pages 137–146, New York, NY, USA, 2011. ACM.

[2] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf. Real-time volume graphics. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[3] S. Guthe and W. Straßer. Real-time decompression and visualization of animated volume data. *Proceedings of the IEEE Visualization Conference*, pages 349–356, 2001.

[4] M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski. Advanced illumination techniques for GPU-based volume raycasting. In *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, pages 2:1–2:166, New York, NY, USA, 2009. ACM.

[5] ImageVis3D. ImageVis3D: A real-time volume rendering tool for large data. scientific computing and imaging institute (sci), 2011. [accessed 29 September 2011].

[6] F. Ino, S. Yoshida, and K. Hagihara. RGBA packing for fast cone beam reconstruction on the GPU. In *, Proceedings of the SPIE Medical Imaging*, 2009.

[7] S. Jeong and A. E. Kaufman. Interactive wireless virtual colonoscopy. *The Visual Computer*, 23(8):545–557, 2007.

[8] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 38–, Washington, DC, USA, 2003. IEEE Computer Society.

[9] F. Lamberti and A. Sanna. A solution for displaying medical data models on mobile devices. In *SEPADS'05*, pages 1–7, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).

[10] M. Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8:29–37, May 1988.

[11] M. Moser and D. Weiskopf. Interactive Volume Rendering on Mobile Devices. In *Workshop on Vision, Modelling, and Visualization VMV '08*, pages 217–226, 2008.

[12] A. Munshi, D. Ginsburg, and D. Shreiner. *OpenGL(R) ES 2.0 Programming Guide*. Addison-Wesley Professional, 1 edition, 2008.

[13] J. Noguera, J. Jiménez, C. Ogáyar, and R. Segura. Volume rendering strategies on mobile devices. In *International Conference on Computer Graphics Theory and Applications (GRAPP 2012). Rome (Italy)*, pages 447–452, 2012.

[14] Power VR. PowerVR Series5 Graphics SGX architecture guide for developers, 2011.

[15] J. Schneider and R. Westermann. Compression domain volume rendering. *Proceedings of the IEEE Visualization Conference*, pages 293–300, 2003.

[16] S. Thelen, J. Meyer, A. Ebert, and H. Hagen. Giga-scale multiresolution volume rendering on distributed display clusters. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6431 LNCS:142–162, 2011.

[17] B. Tomandl, P. Hastreiter, C. Rezk-Salama, K. Engel, T. Ertl, W. Huk, R. Naraghi, O. Ganslandt, C. Nimsky, and K. Eberhardt. Local and remote visualization techniques for interactive direct volume rendering in neuroradiology. *Radiographics*, 21(6):1561–1572, 2001.

[18] L. C. Vera. Volumetric medical images visualization on mobile devices. Master's thesis, Polytechnic University of Catalonia, 2010.

[19] D. Weiskopf. *GPU-based interactive visualization techniques*. Mathematics and visualization. Springer, 2007.

[20] C. Wooley. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, chapter 35, pages 557–571. Addison-Wesley Professional, 2005.

[21] K. Xie, J. Yang, and Y. , Zhu. Real-time visualization of large volume datasets on standard pc hardware. *Computer Methods and Programs in Biomedicine*, 90(2):117–123, 2008.

[22] H. Zhou, H. Qu, Y. Wu, and M. yuen Chan. Volume visualization on mobile devices. In *14th Pacific Conference on Computer Graphics and Applications*, pages 76–84, 2006.