Practical Volume Rendering in mobile devices

Pere Pau Vázquez Alcocer[1]
and Marcos Balsa Rodríguez[2]

[1] UPC, MOVING Graphics group, Spain
www: http://moving.upc.edu/
e-mail: ppau@lsi.upc.edu
[2] CRS4, Visual Computing Group, Italy
www: http://www.crs4.it/vic/
e-mail: mbalsa@crs4.it

Abstract. Volume rendering has been a relevant topic in scientific visualization for the last two decades. A decade ago the exploration of reasonably big volume datasets required costly workstations due to the high processing cost of this kind of visualization. In the last years, a high end PC or laptop was enough to be able to handle medium-sized datasets thanks specially to the fast evolution of GPU hardware. New embedded CPUs that sport powerful graphics chipsets make complex 3D applications feasible in such devices. However, besides the much marketed presentations and all its hype, no real empirical data is usually available that makes comparing absolute and relative capabilities possible. In this paper we analyze current graphics hardware in most high-end Android mobile devices and perform a practical comparison of a well-known GPU-intensive task: volume rendering. We analyze different aspects by implementing three different classical algorithms and show how the current state-of-the art mobile GPUs behave in volume rendering.

1 Introduction

Discrete 3D scalar fields are used in a variety of scientific areas like geophysics, meterology, fluid flow simulations and medicine; its exploration allows scientists to extract different types of relevant information. The most outstanding property of this kind of data is the availability of information in the whole volume, with each space portion having differentiate values. Mesh-based visualization [1] is not well suited to explore all this information since it is typically surface-oriented. This led the development of a new branch of scientific visualization to focus on volume rendering with the objective of enabling exploration of this enormous sources of data. The last two decades volume rendering has been a very active research topic generating many publications covering different issues like illumination, compression or massive model exploration.

One of the main difficulties of volume rendering is the amount of information to deal with. In the 1990s costly workstations were required to work with volume models using software rendering or special-purpose hardware. It was in the later 2000s that interactive volume visualization became possible in high-range desktop and laptop computers by exploiting the texture functionality present in consumer graphics hardware [2].

Nowadays, laptops are being replaced by lighter and smaller embedded devices, like smartphones or tablets, for everyday's work. These devices have become powerful enough to run complex 3D applications previously only available to high end PCs and laptops. Current generation of mobile devices is able to run 3D games with quality comparable to the previous generation of console games.

Many vendors often praise the horsepower of the new CPUs and GPUs sported in mobile phones. However, it is difficult to predict their performance for graphics intense tasks, since it depends on a complex combination of computation power, memory, bandwidth, and several other factors. Therefore, we decided to evaluate the suitability of most modern devices for one well-known GPU-consuming scenario: volume rendering. The contributions introduced in this work are:

– An exhaustive analysis of the most recent mobile platforms and mobile devices currently available in the market.
– An in-depth analysis of the performance of three state-of-the-art volume rendering methods on a subset of the most relevant graphics hardware available in modern mobile devices.

2 Previous Work

2.1 Volume Rendering Algorithms

Volume rendering is a set of techniques used to display a 2D projection of a 3D discretely sampled dataset. These 3D datasets can come from different sources: fluid simulation, geological exploration, medical images or industry object scans. We will focus on medical models. The two most popular volume rendering techniques are RayCasting and Texture Slicing.

RayCasting works by tracing rays from the camera into the volume and solving the rendering integral along this rays. Volume ray casting was introduced by Levoy[3] two decades ago; Krüger[4] et al. presented one of the first GPU implementations one decade ago. Ray casting was done in software for many years due to the lack of hardware support, which was introduced with programmable shader functionality and 3D texture support. A modern GPU implementation of this technique relies on the fragment shader to perform the tracing of rays from eye view into the volume.

Together with ray casting, **texture slicing** is the second most popular technique for GPU based volume rendering. It is an *object-order* approach. The proxy geometry used to render the volume data are 2D slices or quads. The slices are projected onto the image plane and combined according to the composition scheme. Slices can be sorted front-to-back or back-to-front, although probably the most popular is back-to-front order and relying on hardware color blending. Since this technique only relies on standard 2D textures and texture blending, which are available in graphics hardware since many years, it is the most compatible and efficient technique. These techniques, as other *object-order* algorithms, use simpler addressing arithmetics because of working in storage order and so have better performance without complex improvements. These slices can be *object-aligned* or *view-aligned*. Object-aligned slices require having three slice sets in GPU, one for each axis, since we need to render the slice set that is most perpendicular to the view direction. There are graphic glitches when switching from one slice set to another. *View-aligned slices* do not have these problems; however, *view-aligned slices* require 3D texture support. The proxy geometry must be calculated each frame depending on the view position; for this purpose, a bounding box is intersected with planes perpendicular to the view direction and regularly arranged.

2.2 Mobile devices graphics hardware

Mobile devices, especially high-end models, have been typically accompanied by graphics acceleration hardware, only 2D acceleration was supported at first but current devices typically include 2D and 3D acceleration. As of today, it is common in high-end mobile devices to have at least a resolution of 480 pixels width and 800 pixels height, while while next generation will introduce resolutions around 1200x720 (Samsung Galaxy Nexus and most Android tablets) or 2048x1536 (iPad3). This resolution increase comes together with a great increase in graphics hardware performance. This is possible thanks to the powerful CPU and GPU that they all have built-in. All of the devices included in the comparison have support for OpenGL ES 2.0 enabling the use of shaders for graphics programming. There are mainly five dominating architectures in the market (see Table 1):

- Qualcomm. Qualcomm chipsets have been implemented in many devices from a wide range of manufacturer's, with HTC being its most dedicated customer. The SoC solutions provided by Qualcomm come with a graphics solution of its own called Adreno. There are three generations of Adreno GPU's: 200 (Nexus One), 205 (Htc Desire HD) and 220 (HTC Sensation); each generation easily doubles the graphic performance of its antecessor.
- Texas Instruments. TI is one of the most well known embedded device manufacturers and has been present in many Motorola devices and also in recent LG devices (LG Optimus 3D). TI has frequently used the Power SGX 535 and Power SGX 540 GPUs from Imagination Technologies.
- Samsung. In the last years, has also developed a couple of ARM chipsets: the Hummingbird implemented in the Samsung Galaxy S and accompanied by a Power SGX 540 GPU, and the Exynos dual-core implemented in the Samsung Galaxy S2 with the Mali-400MP GPU from ARM.
- NVIDIA. Last year NVIDIA introduced its Tegra 2 platform which is an implementation of ARM's instruction set and accompanied by an Ultra-low voltage (ULV) GeForce graphic chipset also by NVIDIA. This was one of the very first dual-core solutions for mobile devices and is present in the majority of Android tablets sold the past 12 months.

– Apple. Initially integrated chipset solutions from other companies but with the iPhone 4 they started developing their own chipsets, like the A4 in iPhone 4. This ARM processor is complemented with a Power SGX 535 GPU.

Table 1: Comparison of most extended mobile GPU hardware

Model	**MTris/sec**	**MPix/sec**	**3D textures**	**Manufacturer**
Adreno 200	22	133	Yes	Qualcomm
Adreno 205	41	245	Yes	Qualcomm
Adreno 220	88	500	Yes	Qualcomm
Power SGX 535	14	500	No	Imagination Technologies
Power SGX 540	28	1K	No	Imagination Technologies
Power SGX 543	40-200	1K	No	Imagination Technologies
Power SGX 543MP	40-532	1K-16K	No	Imagination Technologies
Mali 400MP	30	300-1K	No	ARM
Tegra2	71	1.2K	No	NVidia

In Table 1 there is a comparison of the most advanced graphic chipsets used in current high-end mobile devices. There are two predominant manufacturers in this table: Qualcomm and its Adreno family of GPUs, and Imagination Technologies with its Power SGX GPUs. Last year, the Tegra2 chipset from NVIDIA has gained importance, specially for being included in almost all new Android tablets. The Mali-400MP GPU is ARM's proposal for their reference design. All the GPUs present in this table offer support for OpenGL ES 2, and so shader programming and state-of-the-art graphics. The only GPUs offering 3D texture support are the ones from Qualcomm: the Adreno family. The numbers in this table are mostly given by the manufacturers and should be taken only as peak values, since they are very conditioned by hardware configuration parameters such as clock frequency. For the sake of comparison, let's say that current generation consoles peak numbers are not too far from those provided by mobile hardware: the XBOX 360 has a peak of 500 million of triangles per second, while the PS3 peak is at 250 million of triangles per second. And for the mobile GPUs let's mention the Power SGX 540, that could reach 90 million of triangles per second by increasing clock's frequency to 400Mhz, and the NVIDIA Tegra 2 with a peak theoretical limit of 71 million of polygons per second. On the other hand, for the purposes of our empirical comparisons, the most relevant numbers are those related to fill rate (millions of pixels per second) because volume rendering applications use little geometry and depend mostly on fragment processing. Comparing typical mobile GPU fill rate of 1 billion of pixels per second with medium range Desktop GPU (like NVIDIA GTX 460) fill rate around 37 billion of pixels per second, shows that there is still an important gap to be filled. It must be taken into account that typical mobile devices have to render 384.000 pixels for a screen of 480x800, while a Full HD desktop monitor would require rendering $2,073.600$ pixels, which is 5.4X the number of pixels in a mobile device. Anyway, including these $5.4X$ factor in the comparison still gives 6.8 times more fill rate to medium-range desktop GPUs.

3 Implementation details

Our application implements two different rendering techniques for volume rendering: object-aligned slices and ray casting. The *object-aligned slices* technique is implemented both for *2D textures* and *3D textures* in order to make the application compatible with more hardware. Not all the techniques work on every device since 3D textures are not common in embedded graphics chipsets, but having these different implementations allows us to compare which one performs better on different hardware. We have also implemented a benchmark thought to be executed on many different devices with heterogeneous hardware configurations in order to compare performance.

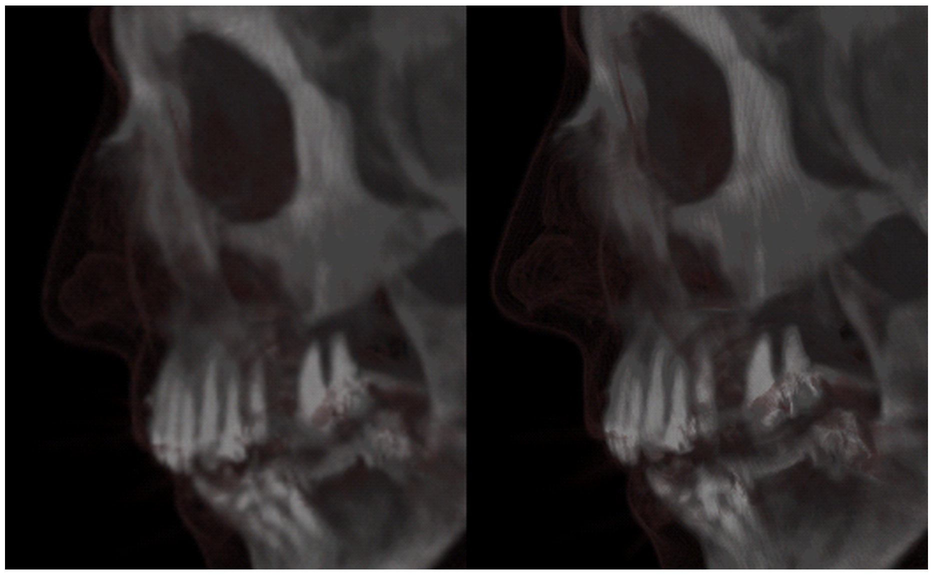

Fig. 1: These images compare a detail in two renditions of the CT head dataset at half viewport resolution (left) and full viewport resolution (right)

Since the frame rates are not high, as we will show later, we need to add some improvements in order to make the application interactive. Being the application so pixel intensive, we took to two different approaches for level of detail: reducing the number of slices and reducing the viewport resolution. When low viewport resolution is enabled, static renders are also done in half resolution; thanks to linear interpolation using half the resolution of the viewport still give good visual quality slightly alleviating the aliasing (see Figure 1).

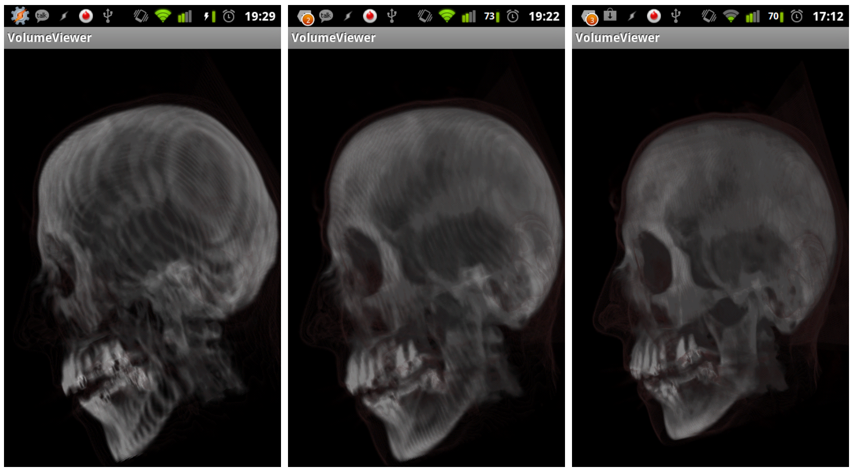

Fig. 2: These images show renditions at full viewport size of the CT head dataset (with dimensions 256x256x113) using 64, 128, and 256 slices, respectively

On the other hand, reducing the number of slices without using lower resolution viewport produces more noticeable artifacts (see Figure 2). For this reason, we propose using the maximum number of slices to match dataset dimensions and simply enable low viewport resolution to improve interaction. We offer the user the option to define a lower slice count for rendering while there is user interaction for slowest devices, but in our tests it has only been needed for ray casting where the hardware still performs at low frame rates.

We have also taken care of the usability of our approach. For volume models, the definition of a transfer function is a very important step, since it determines which information is visible and how. We implemented a transfer function editor tailored to small screens with numerous visual feedbacks for selection, such as the selection highlighting in yellow, or the mini-zoom tool tailored to perform fine selection whilst avoiding the problem of finger occlusion (see Figure 3).

4 Results

We have performed a variety of performance tests with different configurations to be able to extract meaningful information about different hardware restrictions:

– **Sampling resolution**. We have run benchmarks with different volume sampling frequencies (number of slices for slice-based renderers or number of samples per voxel for the ray cast renderer) to analyze the impact of voxel

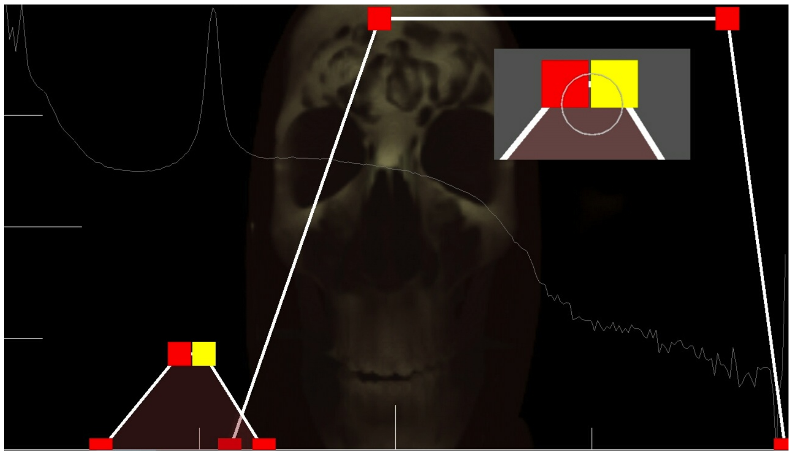

Fig. 3: Edition of the transfer function with the mini-zoom helper. This function defines two sub-functions associating transparency and color to the scalar values

color composition on performance. We have run the tests with the Engine dataset with dimensions 256x256x256 because most of the devices tested haven't been able to load larger datasets. Only for the 3D texture slice renderer and for the ray caster we have run the benchmark with the Sheep dataset with dimensions 352x352x256.

- **Viewport resolution**. We have run benchmarks in full resolution (480x800) and half resolution (240x400) to analyze the effect of fill rate.

The benchmarks consist of a series of camera positions that are rendered consecutively. These camera positions start very close to the viewer with the volume covering the full viewport and gets farther and farther progressively while rotating exactly five times 360 degrees until the volume only covers about 1/8 of the screen. This way we get an averaged frame time from all the views of the volume covering the full viewport and only covering a small portion.

We have defined four different qualities based on sampling frequency. For slice-based renderers, quality 0 uses at most 64 slices per axis, doubling the slice count until quality 3 where 512 slices per axis is the limit (depending on the dataset). Quality is defined as the number of samples per voxel for the ray casting renderer, with 0 being 0.25 samples per voxel (taking into account only 1 of every 4 voxel) and 3 being 1 sample per voxel. It must be noted that for the 2D texture slice renderer we resample the textures in order to reduce the texture data size. This allows us to load bigger volumes although the rendering will be in lower resolutions (ie. the slices in a 512^3 dataset in quality 2 would be resampled to 256^3). The maximum effective volume size we have been able to load in most devices is the 256^3 *Engine* dataset for this renderer; it has the big disadvantage of requiring 3 slice sets, one for each axis, and so uses three times more GPU memory. For the 3D texture slice renderer and the ray cast renderer the limits are imposed by the drivers and the largest volume dataset that we have been able to load is the 352x352x256 *Sheep* dataset.

Table 2: Benchmark results of the implemented volume renderers on different mobile devices in full viewport resolution and half viewport resolution. Quality is the number of slices [64, 128, 256, 512] for slice-based renderers and the number of samples per voxel [0.25, 0.5, 0.75, 1] for the ray casting renderer. The values in the table are frames per second

		high resolution				low resolution			
		Galaxy S	Advent Vega	HTC desire	HTC desire Z	Galaxy S	Advent Vega	HTC desire	HTC desire Z
	Qual.								
slices 2d	0	6.28	7.41	5.75	13.25	11.49	22.73	15.15	34.48
	1	3.24	3.65	3.08	7.35	6.85	12.66		22.47
	2	1.91	1.92	2.2	5.38	3.64	7.14	5.92	9.95
	3								
slices 3d	0			5.08	10.31			6.76	11.24
	1			2.65	5.15			4.37	6.62
	2			1.84	2.33			2.33	3.71
	3			1.81	1.63			2.43	2.94
ray cast	0			0.42	2.06			1.71	5.38
	1				1.54			0.95	4.02
	2				0.96			0.65	3.06
	3				0.77				1.96
GPU		Power SGX 540	Tegra 2	Adreno 200	Adreno 205	Power SGX 540	Tegra 2	Adreno 200	Adreno 205

Table 2 shows the frame rate values obtained from running the benchmark on different devices. The 256^3 Engine model has been used for all the benchmarks. All these devices have a screen resolution of 480x800 and the results are for low (half the viewport resolution) and for high resolution (full viewport resolution). From this table, we can infer that the *Samsung Galaxy S*, with a performance boost of $2X$, is less affected by reducing the resolution than the HTC desire, which gets a boost of $3X$. The *Adreno 205* GPU in the *HTC Desire Z* and the *Tegra 2* in the *Advent Vega* show the same $3X$ performance boost with half resolution. The *Advent Vega* has not achieved the results that one could expect from the *Tegra 2* chipset (while the *LG Optimus 2X* also including the *Tegra 2* performs significantly better, as will be seen later).

4.1 Device performance comparison

We have run our benchmarks on many devices to illustrate the results of the 2D texture slice renderer, which is the most compatible approach; this will give us a good overview of current mobile GPUs performance.

In Figure 4 we can see benchmark results for all the devices we have tested with the slice renderer based on 2D textures. There are four differentiated groups:

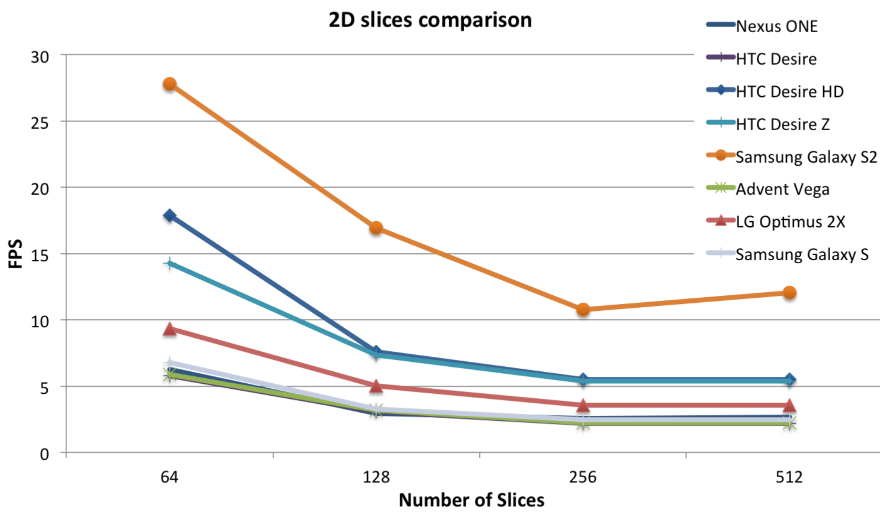

Fig. 4: Benchmark results of various devices for the 2D texture slice renderer. Lowest lines show Nexus One, Samsung Galaxy S with previous generation single core hardware and the Advent Vega, a low-cost Tegra 2 device

- The *Samsung Galaxy S2*, with a dual-core processor including the Mali-400MP GPU, achieved the best performance by almost doubling its nearest competitor. This is one of the latest devices in the market at the time of writing. On the other hand, they have not support for 3D textures.
- The *HTC Desire HD/Z* are two recent devices from HTC; they implement the second generation of Adreno processors, the Adreno 205. This GPU family is the only one that we know to have 3D texture support. Its performance is in the second range quite below the Galaxy S2.
- The *LG Optimus 2X*, with a dual-core CPU based on NVIDIA Tegra 2 platform, is half way between the Qualcomm first and second generations of GPUs. The Tegra 2 platform was expected to perform much better, but seems that for our benchmark implementation this is not the case.
- The *Nexus ONE, HTC Desire, Samsung Galaxy S and the Advent Vega* are in the last group. The first two integrate the Adreno 200 from the first generation of Qualcomm chipsets, while the Samsung Galaxy S sports a powerful Power SGX 540 from Imagination. The Advent Vega seems to lack some driver optimizations since the same platform in the Optimus 2X has performed significantly better. Taking the Advent Vega apart, this last group if composed of some of the most extended single-core Android devices.

4.2 Comparing the volume rendering implementations

We see how the slice-based renderers have much better performance than the ray casting; this is mainly because the work done in the fragment shader is much lighter and most of the work is done in the composition phase performed just after that using the typical hardware pipeline which is much more optimized. For the 2D texture slice renderer we have used the 256^3 Engine dataset and so quality 2 and 3 have the same performance. The trilinear filtering used by 3D textures is one of the reasons of the performance gap between the 2D and 3D slice renderer, trading visual quality for performance. For the ray casting renderer the intensive use of fragment shaders takes the GPU to its limits.

Mobile GPUs are not yet as capable as their desktop and laptop counterparts due mainly to low graphic unit count and no dedicated graphic memory. Latest mobile GPUs typically have between 4 and 12 processing units or shaders, depending on them having unified shader architecture or implementing Vertex/Fragment shaders, and shared system memory with some fraction of this memory reserved for the GPU. On the other hand, latest mobile devices sporting these GPUs also include high resolution screens (ranging from 800x400 to 1280x720 and 5" to 10"). These two facts: low processing unit count with no dedicated memory and high resolution screens introduce a big bottleneck into the fragment shader phase. We have seen that typically using partial resolution is enough for giving good results in such small screens, specially while interacting, mainly because human eye is not able to exploit this high resolution at the typical usage distance (around 15-30 cm).

5 Conclusions and Future Work

We have achieved interactive frame rates on most high-end mobile devices currently available. We also implemented a transfer function editor that is specially designed for small screens. However, many aspects could be improved on both sides. The object-aligned slices approach produces disturbing graphic glitches when changing the point of view from one axis to another; also there are noticeable artifacts when looking at steep angles. For the 2D texture slice renderer, requiring three slice sets is unavoidable but by using the volume rendering schema presented by Krüger[5] the graphic glitches produces by slice set changes should be unnoticeable. Also the usage of multiple textures per slice and implementing trilinear filtering on the shader should improve visual quality. All these extensions will add a considerable CPU and GPU cost. For the 3D texture slice renderer the most noticeable problem is due to the object-aligned slices again, requiring high number of slices to produce a high quality view. In this case, implementing view-aligned slices would give much better results at the expenses of some performance loss due to the calculation of new slices each frame. The ray casting renderer we have implemented is very basic and can be greatly improved both for quality and performance. Using too few samples produces many artifacts, but this is unavoidable due to the low performance of this technique on current hardware. To improve performance, we could use empty-space skipping [6] to avoid sampling empty regions at the expenses of using another low resolution 3D texture and one extra texture access at each sample point. However, these are improvements we plan for next generation devices. At the moment of the implementation, the most advanced devices and were LG Optimus 2x and Samsung Galaxy SII. All the implemented techniques could be greatly improved by adding shadows. However, this would add quite many calculations on the fragment shader and could not be feasible.

Acknowledgments

This project has been supported by TIN2010-20590-C02-01 Project of the Spanish Government and People Programme (Marie Curie Actions) of the EU's 7th Framework Programme FP7/2007-2013/ under REA grant agreement n290227.

References

1. Akenine-Moller, T., Haines, E., Hoffman, N.: Real-Time Rendering, Third Edition. A K Peters (2008)
2. Engel, K., Kraus, M., Ertl, T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware. HWWS '01, ACM (2001) 9–16
3. Levoy, M.: Display of surfaces from volume data. IEEE Comput. Graph. Appl. **8** (1988) 29–37
4. Krüger, J., Westermann, R.: Acceleration techniques for gpu-based volume rendering. In: Proceedings IEEE Visualization 2003. (2003)
5. Krüger, J.: A new sampling scheme for slice based volume rendering. In: Volume Graphics. (2010) 1–4
6. Li, W., Mueller, K., Kaufman, A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In: In Proc. IEEE Visualization 2003. (2003) 317–324