Progressive Ray Casting for Volumetric Models on Mobile Devices

Jesús Díaz-García[a], Pere Brunet[a], Isabel Navazo[a], Pere-Pau Vázquez[a]

[a] *Universitat Politècnica de Catalunya*

Abstract

Mobile devices have experienced an incredible market penetration in the last decade. Currently, medium to premium smartphones are relatively affordable devices. With the increase in screen size and resolution, together with the improvements in performance of mobile CPUs and GPUs, more tasks have become possible. In this paper we explore the rendering of medium to large volumetric models on mobile and low performance devices in general. To do so, we present a progressive ray casting method that is able to obtain interactive frame rates and high quality results for models that not long ago were only supported by desktop computers.

Keywords: Direct Volume Rendering, Progressive Ray Casting, Mobile Devices

1. Introduction

In the last years, thanks to their ubiquity and increasing computational power, laptops, smartphones, tablets and mobile devices in general, have become more and more suitable for applications that require high quality visualization of volume data in real time. Medical visualization is one of the most important application fields. Unfortunately, despite the increase in computational power, visual quality and storage capacity, certain tasks such as high quality visualization and interactive exploration of medical models still represent a challenging problem for this kind of hardware.

There has also been an increase of the published research about how to deal with visualization on mobile devices efficiently [1, 2]. However, many contributions have been striving to address some limitations of hardware constraints that are no longer a problem. A clear example of this is the recent addition of 3D textures into the OpenGL ES standard from version 3.0. Thus, we base our work on the usage of 3D textures along with GPU ray casting, which is the state-of-the-art technique used for volume visualization.

Previous experiments have shown that, even though big models might fit into the memory of such GPUs, the rendering performance achieved by mobile devices is still not enough. Usually, the visualization of models with larger resolutions ($\geq 512^3$) that still fit in the graphics memory of mobile devices, achieves low frame rates which are far from being interactive and prevent the user from experiencing a smooth exploration of the model.

An easy way to gain interactivity when dealing with large models is sacrificing visualization quality by reducing both the displayed dataset and the viewport resolution. Each time the user stops interacting, it is desirable to obtain a high quality image of the resulting point of view. However, in order to achieve a high resolution image, even the process of rendering a single frame is enough for a standard mobile device to stall until the GPU is free from rendering tasks and can return the control to the application. Such stall, if long enough, leads to the operative system killing the application to guarantee that the device is not blocked. In [3, 4, 5], the authors notice this fact as a consequence of executing a high number of instructions in shader code. We noticed a similar behavior for camera views where rays performed a large number of samples, making the application crash after being blocked by rendering tasks during a certain amount of time (between 2 and 3 seconds in our experiments). Although this issue could be solved by detaching the rendering process from the main GUI thread, this is actually not desirable in some scenarios such as in medical applications, where high resolution results are expected to be provided as soon as possible. By offloading the rendering task from the GUI thread, application crashes can be avoided, yet long rendering processes will still provide delayed results, and this lack of feedback is likely to cause confusion.

In this paper, we propose a solution that uses progressive GPU-aided ray casting algorithms to generate high quality renderings. This strategy prevents blocking and provides interactivity, distributing the rendering task over subsequent frames after every user interaction. The main contributions of our approach are:

- A multiresolution, scalable rendering scheme for volume models on mobile devices that uses a low resolution model during user interaction and a high resolution dataset for quality visualization when the camera stops.
- A strategy pattern for incremental rendering that provides a smooth transition from the low resolution visualization to the high resolution visualization, preventing blocking to avoid undesirable application aborting and allowing for smooth interactions at any time.
- The proposal of two new progressive ray casting methods

Email addresses: jesusdz@cs.upc.edu (Jesús Díaz-García), pere@cs.upc.edu (Pere Brunet), isabel@cs.upc.edu (Isabel Navazo), ppau@cs.upc.edu (Pere-Pau Vázquez)

that fulfill the aforesaid goals, and their analysis, discussion, and a comparison with other existing techniques.

The rest of the paper is structured as follows: Section 2 presents the previous work carried out in this field. Then, an overview of the architecture we have designed is explained in Section 3 and the detailed algorithms we use for progressive ray casting are presented in Section 4. Section 5 shows the results we have achieved, and then some conclusions and lines for future work are discussed in Section 6.

2. Previous Work

Nowadays, GPU ray casting is the state of the art technique in Volume Rendering [6]. Current desktop systems allow the rendering of huge models in real time, typically using compression and/or out-of-core techniques [7, 8].

Several publications deal with volume rendering of large scalar fields. For instance, Crassin et al. [9] propose an approach based on an adaptive data representation, depending on the current view and occlusion information directly extracted from the rendering algorithm. In the same line, Fogal et al. [10] introduce an out-of-core, ray-guided GPU volume renderer that scales to large data and provide an evaluation and discussion of the trade-offs inherent to this kind of application. Both of these techniques rely on the fact that not all the information needs to be visualized at its maximum resolution (mainly due to camera perspective and occlusion). However, medical applications are meant to provide maximum quality without sacrificing interactivity.

Callahan and Silva [11] present an image-space acceleration technique based on a bilateral upsampling filter to improve the quality of low-resolution ray casting images of unstructured volumes. They are able to preserve features at the edges of the models thanks to a guidance image obtained from the unstructured grid. Unfortunately, structured grids (which are the most typical format for medical images), cannot benefit from this technique straightforwardly.

Although not explicitly working on mobile platforms, the following publications, target the issue of incremental ray casting. Levoy [12] introduced an incremental way of performing volume ray casting based on an adaptive image space subdivision. In [13], Kratz et al. improved Levoy's approach by introducing an error estimator from the field of finite element methods. In the same line, Frey et al. [14] presented a scheme for progressive rendering that adapts to different changes during data exploration. They demonstrate an automatic parameter optimization scheme using a video metric to optimize their frame control. These techniques are mainly focusing on quality metrics to lead the progressive refinement algorithm. However, the way they distribute rays is not tailored to improve performance, as we will discuss later in Section 4.3.

Since mobile platforms are ubiquitous nowadays, the interest in using mobile devices for rendering volumetric models, especially medical datasets, is growing [15, 16]. Several previous approaches have addressed volumetric models on mobile devices using two different strategies: server dependent methods and local methods.

Server dependent methods that heavily rely on the server side to perform all power consuming operations are called *thin client* architectures. Following this scheme, Lamberti et al. [17] communicate rotation and translation commands from client devices to the server, and obtain an MPEG video stream with the rendered results of medical images as a response. In [18], Hachaj et al. propose a similar solution also based on *thin clients*, and Gutenko et al. [19] use a more efficient and modern video codec (H264) to encode the video stream. However, these methods have strong connection bandwidth requirements that we are interested in getting rid of.

Balanced solutions distribute the tasks between the server side and the mobile device. In this line, Campoalegre et al. [20] perform a block-based transfer function aware compression of the target dataset, and are able to transmit the desired regions of interest to support adaptive ray casting on the client side.

Other server dependent methods that take more advantage of client desktop machines and hand-held devices are called *fat distribution* schemes by some researchers [16], and they mainly rely on the server to provide the datasets after performing some expensive pre-processing tasks, but produce the renderings locally. For instance, Congote et al. [21] presented a platform implementing this kind of architecture by means of the WebGL standard. Mobeen et al. [4, 5] also developed various algorithms that perform a single-pass ray casting for the efficient visualization of medical models based on WebGL [22, 23]. A detail-on-demand scheme is presented by Schultz and Bailey [24], where they allow the user to explore the entire dataset at its original resolution while simultaneously constraining the 3D texture size so that it doesn't exceed the GPU capabilities of the portable device.

Finally, local rendering methods allow the visualization on mobile devices with no need of network connectivity. 3D textures have been widely available in mobile GPUs just recently, so previous methods for rendering volumetric models have relied on 2D texture stacks or tiled 2D textures emulating 3D textures. Among others, Fogal et al. [1] and Noon et al. [3] have developed tools using stacks of 2D textures representing the 3D volume. Congote et al. [21], Noguera et al. [25, 2, 26] and Movania et al [27], on the other hand, emulate 3D textures by using a mosaic layout of its slices within a set of 2D textures. More recently, when 3D textures have become widely available, both slicing and ray casting algorithms have been used. Balsa et al. [28] presented a practical comparison of volume rendering using several devices and algorithms, including ray casting with the use of 3D textures, which was far from interactive at that time. Also using 3D textures, Xin and Wong [29], presented an intuitive framework for volume data exploration, although they don't work with datasets of resolutions higher than 128^3. Furthermore, Schultz and Bailey [24] develop a multiresolution algorithm based on the use of a detail-on-demand subvolume selection with 3D textures.

Nowadays, GPUs in hand-held devices are more capable, so focusing on *fat* and local rendering approaches by implementing the ray casting task on mobile phones seems more feasible.

However, porting volume rendering to mobile devices may be challenged by three main limitations: GPU capabilities (as they could not provide the proper features to deal with the algorithms used to visualize volumetric models), RAM size (models might not fit in main memory), and GPU horsepower (even though models might fit the GPU memory, the frame rate achieved could be inefficient to adequately support interactivity). In this paper, we mainly concentrate on the latter limitation.

3. Overview

Our motivation is to address the problem of volume rendering of medical data on mobile and low performance devices through the development of a system that fulfills the requirements of medical experts, including being able to interact with large volume models, usually involving dataset resolutions larger than 512^3. There is an increasing interest in exploring these volume models on mobile devices at full resolution. Furthermore, an essential prerequisite is that the application must achieve interactive frame rates even with large models, so the system must not stall or have performance drops that hinder the user interaction.

We use GPU-aided ray casting to perform direct volume rendering (see Figure 1), as it is the state of the art technique for the task [6]. Volume ray casting is an image-based technique that casts a ray from each pixel of the final image and computes, along each ray, the accumulated color by evaluating samples in the volume dataset. A volume dataset typically consists of scalar values uniformly sampled in a finite three dimensional space, forming a regular grid, as is the case with medical images. More precisely, the algorithm proceeds by sampling the volume data at regular intervals along each ray. Those sampled values are transformed into color by means of a transfer function that maps intensity values to RGBA data. Ray casting is a technique perfectly suited to be implemented on GPUs due to its highly parallelizable nature (the rendering integral along rays can be computed separately by fragment processors). However, it should be emphasized that its complexity increases rapidly with the resolution of the volume data being sampled, especially if the algorithm requires the computation of gradients to perform good quality shading, which usually requires performing 6 extra texture lookups at each ray sample. Therefore, large datasets imply costly computations of the ray integral, which implies a bottleneck in the fragment shader performing that calculation.

Our implementation of the ray casting algorithm uses several methods that help improving visual quality. Pre-integrated transfer functions [30] are used in order to avoid undesired wood-grain artifacts without sacrificing performance. In addition, we perform downsampling to achieve a low resolution dataset that allows interactive exploration and also whenever the original resolution dataset does not fit the GPU memory. We use a feature-preserving downsampling filter [31] that is able to preserve important features typically lost during the downsampling process. Finally, we use Adaptive Transfer Functions [32] to visualize the coarser levels with higher accuracy.

Furthermore, whenever possible we have incorporated acceleration techniques such as Empty Space Skipping (ESS) and Early Ray Termination (ERT) described by Krüger and Westermann [33]. We perform ESS by means of starting the ray sampling at the boundaries of a proxy geometry that, given a certain transfer function, roughly adjusts to the non-transparent regions of the volume model (see Figure 2). This way, the effective sampling space along rays is allowed to start later and finish before, discarding transparent regions. The second acceleration technique, ERT, finishes ray traversals whenever the computed color is considered to be opaque.

The standard way used by medical experts to inspect medical images is based on orthographic projections. For this reason, we use orthographic cameras for all the implementations in this paper.

Since our goal involves implementing a scalable system that is able to perform interactive high resolution ray casting of large models, we propose a framework based on multiresolution. Our solution uses a lower resolution dataset for the visualization while the user is exploring the model, which ensures interactive frame rates, and a high resolution dataset along with a progressive refinement algorithm for high quality rendering of the desired regions of interest after each user interaction.

The usage of a progressive rendering algorithm ensures that by splitting the ray casting into several frames, the control of execution is returned back to the application loop more frequently, so that it cannot stall for long periods of time, allowing the user to start new interactions at any time, even before the progressive rendering has finished (thus canceling the process).

Based on this general strategy, we propose two different approaches, depicted in Figures 3 and 5. Both share the same structure: during user interaction, rendering is performed using low resolution ray casting (top row). Every time the user stops at a certain view, the progressive high resolution ray casting starts so that the static image of the selected view evolves smoothly from the low resolution ray casting result to the full resolution image.

The main differences between both strategies are the way the high resolution images are produced. In one case, the final image is obtained by rendering the high resolution dataset in separated slabs in front-to-back order (we call this technique Front-to-back Slabs, or *FBSlabs*). The second strategy, on the contrary, splits the viewport into several tiles and sorts them by cost in order to group them into batches of similar cost that can be efficiently rendered at each frame until a certain time budget is reached (we refer to this technique as Sorted Tiles, or *STiles*).

4. Progressive ray casting strategies

In this Section we present details on each of the two proposals for incremental rendering: *FBSlabs* and *STiles*.

4.1. FBSlabs *(Front-to-back slabs)*

This progressive algorithm splits each ray into several segments of a fixed length and then starts rendering those segments in front to back order over subsequent frames after the user finishes interacting. The algorithm renders the incremental high

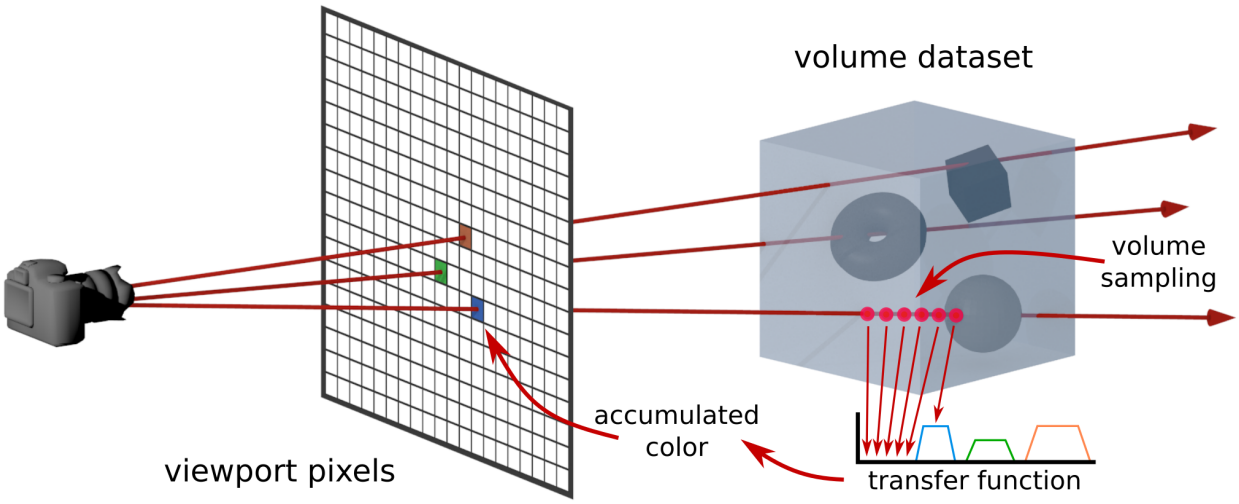

Figure 1: Overview of the ray casting algorithm. A ray emerging from the camera origin is generated for each pixel of the viewport passing through its center. Segments of these rays intersecting the volume dataset, are in turn evaluated at several positions separated by regular intervals, obtaining scalar data from the dataset and transforming it to color values accumulated over the ray traversal. Usually, several extra texture lookups are necessary at each sampling position to compute the local gradient in order to calculate proper shading.

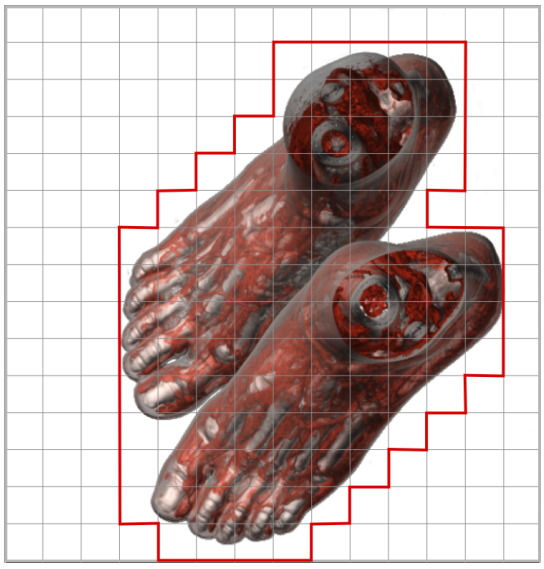

Figure 2: The proxy geometry bounding this volume model is shown in red (in 2D). We subdivide the bounding box of the volume model in a grid and generate a mesh that envelopes those grid cells containing non-transparent data. We use the proxy geometry in order to perform empty space skipping, allowing rays to effectively start where non-transparent data is found, and finishing wherever there is only transparent data remaining.

resolution results into a texture T_{high}, the low resolution results into another texture T_{low} and finally composites both textures to achieve the final image at each frame. These are the main steps followed by this algorithm:

1. **Low Resolution Ray Casting (during user interaction)**
   - The ray casting color is stored in a 2D texture
2. **Progressive High Resolution Ray Casting**
   (a) A 2D texture T_{high} is cleared
   (b) The first sampling position for each ray (one per viewport pixel) is placed at the entry point on the proxy geometry bounding the volume model
   (c) A fixed number of ray casting steps are performed advancing over each ray (rendering a non-regular slab perpendicular to the viewing direction), and the resulting color is composited with the previous high resolution partial result in T_{high}
   (d) The remaining part of the volume is rendered with low resolution ray casting, starting at the sample position where the previous step finished, and then stored into T_{low}
   (e) The current image is generated by compositing T_{high} on top of T_{low} with alpha blending
   (f) In the next frame, a frame counter is increased and the process resumes ray casting from (c) until the sampling positions exceed the volume boundaries

In Figure 3, step 1 indicates that the low resolution rendering is generated by a standard ray casting algorithm, with no modifications, into a 2D texture T_{low}.

Then, in step 2 of Figure 3, a chain of partial ray castings is performed in separated slabs to render the high resolution dataset progressively. A 2D texture T_{high} is used to store the progressive state of the high resolution rendering process. To

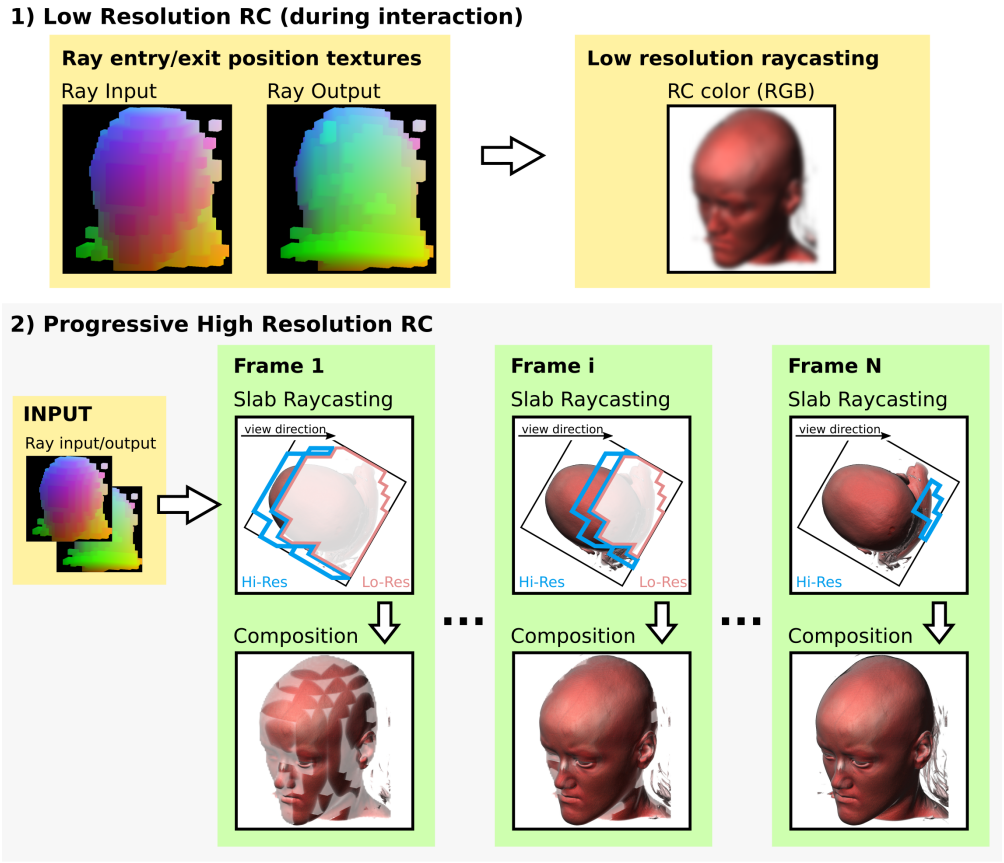

Figure 3: Schematic overview of the *FBSlabs* algorithm. The first step (1) of the figure depicts the initial low resolution standard ray casting performed while the user is moving the camera. Each time the interaction stops (2), the high resolution image is incrementally composited by rendering slabs in front-to-back order, one at a time every frame. Then, at each frame, this high resolution image is composited on top of the remaining part of the model rendered at low resolution.

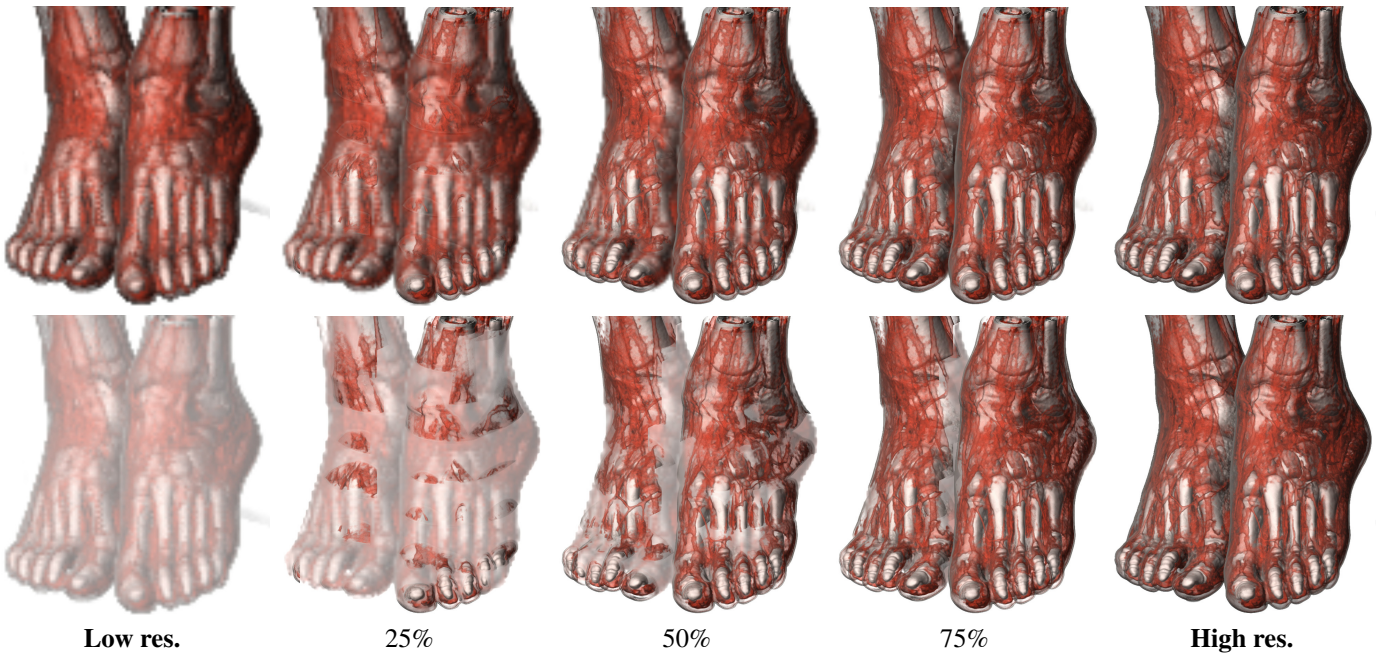

Figure 4: Vix dataset ($512^2 \times 250$ high res., $128^2 \times 63$ low res.). This sequence of images shows the transition effect between the low resolution and the high resolution ray castings obtained by the *FBSlabs* method. The top row shows the renderings as shown in the application, whereas in the bottom row, the low resolution part of the same images is lightened in order to reveal the updated portions of the image more clearly. See the accompanying video for a more clear example of this transition.

start the process, in the first frame after the user interaction finished, the initial segments of all rays emerging from the viewport pixels are rendered in T_{high}. Those ray segments start at the entry points on the proxy geometry, and perform a fixed number of samples ($N = 40$ in our case) in each ray casting frame, making each slab have a fixed thickness. During the next frames, the same slab rendering process is repeated. At each frame, in order to resume the high resolution ray casting where the previous frame finished, we only need to know the current frame counter (number of frames since the progressive ray casting started), as each slab is rendered with a fixed number of samples and a constant sampling space between them. Note that the camera is configured to perform an orthographic projection of the scene, as commonly used in medicine. This way the generated slabs remain piecewise planar as they originate from the proxy geometry. A perspective projection could be used otherwise without causing any trouble, this way leading to pseudo-spherical slabs as they get far from the starting point at the proxy geometry. The blending state is configured to add color in a front to back order in order to update T_{high} with each rendered slab. In T_{low}, the remaining part of the ray casting is computed at low resolution, which implies almost no penalty in time. At each frame, the resulting partial image T_{high} is composited over the low resolution image T_{low} using alpha blending. The high resolution ray casting is completely finished whenever all the ray segments rendered exit the proxy geometry. We conservatively approximate this moment by repeating this iterative process until the computed rays are longer than the diagonal of the volume bounding box. Figure 4-top shows the transition effect of this technique (in Figure 4-bottom, color is modified to better perceive the boundary between the low resolution and the high resolution rendered parts).

4.2. STiles *(Sorted tiles)*

This progressive ray casting algorithm first decomposes the high resolution image space into square blocks of pixels (tiles) and then renders them progressively over subsequent frames (see Figure 5). The rationale behind this method comes from the tile-based behaviour of the GPU rasterizer and cache usage. Analogously to *FBSlabs*, the low resolution rendering is stored into a low resolution texture T_{low} and the high resolution results are incrementally rendered into a high resolution texture T_{high}. The algorithm pipeline proceeds through the following steps:

1. **Low Resolution Ray Casting (during user interaction)**
   - The ray casting color is stored in a 2D texture T_{low}
   - The ray cost (number of ray samples) is stored in the alpha channel of T_{low}
2. **Tile Sorting (once after interaction finished)**
   - Once the user interaction stops, the screen space is split into tiles, and then, tiles are sorted by cost (using a series of compute shaders), generating two correlation maps that allow converting between unsorted and sorted tile coordinates.
3. **Progressive High Resolution Ray Casting**
   (a) The high resolution ray casting of a few tiles (rendered in order) is performed, until a fixed time budget is reached
   (b) The current image is composited, selecting either the high resolution pixels from T_{high} when already computed, or the low resolution ones from T_{low} otherwise
   (c) In the next frame, the process is resumed from (a) until all the tiles are rendered

During user interaction, a standard low resolution ray casting for interactive rendering is performed in a fragment shader (see step 1 of Figure 5). At each pixel, together with the low resolution color in the RGB channels, the number of ray samples is stored in the alpha channel of the output texture T_{low} as an estimation of the ray cost. This ray cost approximation is crucial for the main goal of the algorithm.

The second step starts once the user stops interacting. The viewport is then divided into small tiles, and these tiles are in turn sorted according to the ray cost hint provided by the previous stage (see step 2 of Figure 5), by means of a few compute shaders that implement a GPU radix sort algorithm [34]. The sorting pipeline proceeds in three steps, each one carried out by a compute shader: *i)* Group counting, *ii)* Group offset setting and *iii)* Sorting. During the first step *i)*, the tiles are classified into groups depending on its cost, so that we finally have a counter of the number of tiles belonging to each group. We consider the cost of a tile to be the number of ray samples (previously stored in the alpha channel of T_{low}) at the center of the tile. The second stage *ii)* scans these counters to establish an offset for each group of tiles, so that they can be later placed in an output texture without overlapping. Finally, the third and last step *iii)* proceeds by sorting tiles, placing them into the right position defined by their group offset, depending on their cost. The actual outputs of this compute shader are two texture maps that allow translating from sorted to unsorted tile coordinates and vice versa.

Finally, the last step of *STiles* corresponds to the progressive ray casting, carried out again by a fragment shader. It renders a variable number N of screen tiles, in order, in a separated 2D texture alias of sorted tiles. Thanks to the coordinate maps produced in the sorting stage, the tiles can be rendered in strict order. The variable number of tiles depends on a fixed time budget (0.1 seconds in our case). The algorithm proceeds by rendering a window of N tiles. After rendering these N tiles, the elapsed time is measured in order to know if the time budget has been exceeded, and in this case, it does not render any more tiles during this frame, otherwise it renders N more tiles until the time budget is reached. The final image is composited by either selecting, for each pixel, the high resolution ray casting color if available (again, using the coordinate maps produced in the sorting stage to know its position in the sorted tiles texture), or the low resolution ray casting color otherwise. This last step is repeated in successive frames, rendering as many tiles as possible without exceeding the fixed time budget, until the whole high resolution ray casting image has completely substituted the low resolution one (see step 3 of Figure 5). Fig-

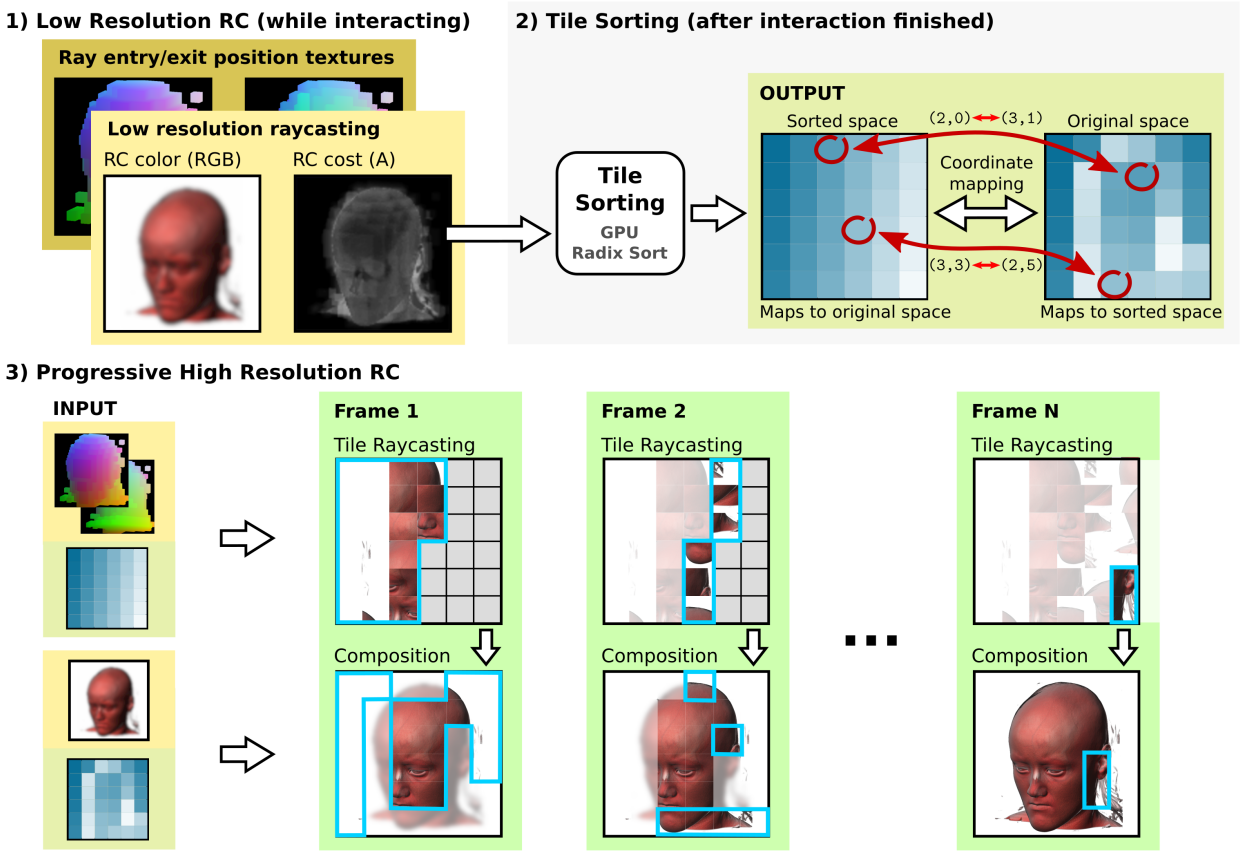

Figure 5: Schematic overview of the *STiles* algorithm. The first step (1) of the figure depicts the initial low resolution standard ray casting performed while the user is moving the camera (the ray cost hint is stored in the alpha channel). Each time the interaction stops (2), the screen space is split into tiles and sorted according this ray cost hint, and two coordinate maps that are able to convert from one space to another are generated. The incremental rendering (3) proceeds frame by frame, rendering tiles in order and compositing the final image by selecting pixels either from the low resolution texture or from the high resolution tiled texture.

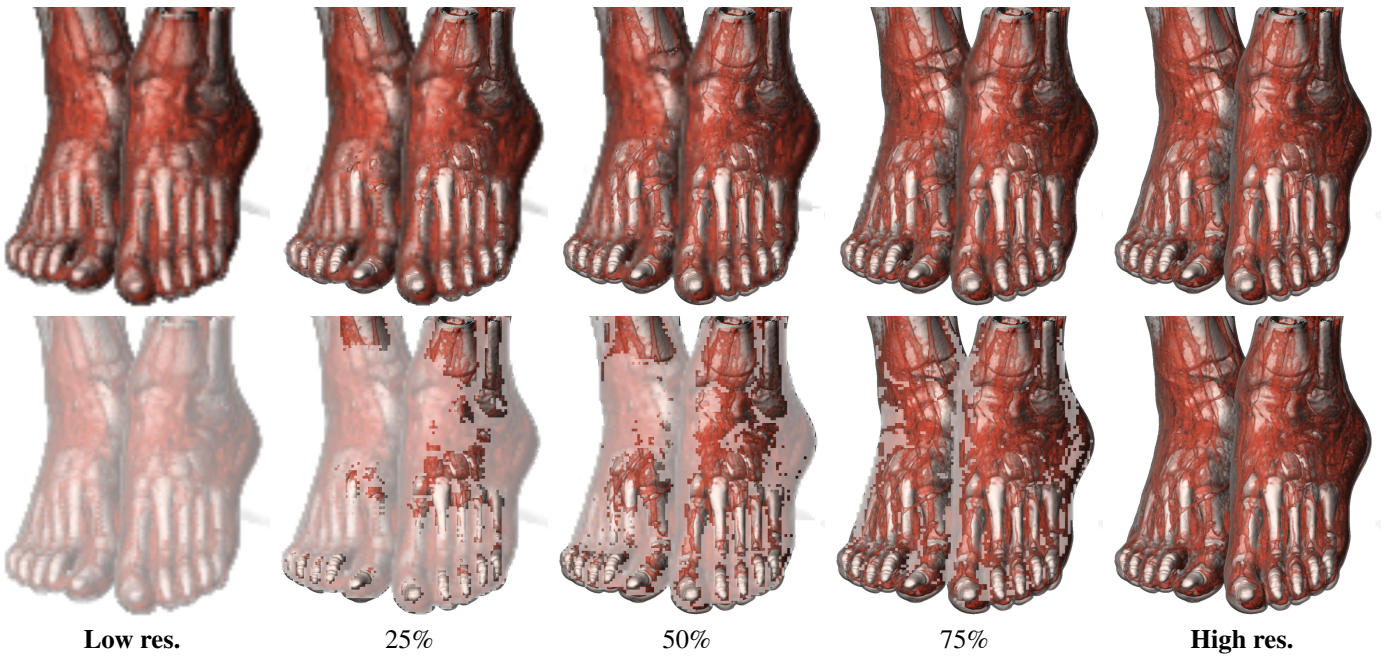

Figure 6: Vix dataset ($512^2 \times 250$ high res., $128^2 \times 63$ low res.). This sequence of images shows the transition effect between the low resolution and the high resolution ray castings obtained by the *STiles* method. The top row shows the renderings as shown in the application, whereas in the bottom row, the low resolution part of the same images is lightened in order to reveal the order in which the image is updated. See the accompanying video for a more clear example of this transition.

ure 6-top shows the transition effect of this technique (in Figure 6-bottom, color is modified to better perceive the boundary between the low resolution and the high resolution pixels).

4.3. Discussion

We have described two different strategies based on GPU ray casting for the incremental rendering of high resolution volume datasets. Both are fast and complete the rendering of the final image quick enough to be considered good candidates for our purposes. One of the main strengths of the *FBSlabs* method is its low requirements regarding GPU specifications and OpenGL version. As our architecture is based on the use of 3D textures, a minimum version of OpenGL ES 3.0 is needed, but a different implementation that makes use of 2D textures to store the dataset in GPU memory could support lower versions of OpenGL. In this sense, *STiles* has stricter requirements, demanding a minimum version of OpenGL ES 3.1 on mobile devices, due to the usage of compute shaders, which were not available in previous versions. For this reason, not only old graphics chips, but also WebGL platforms, are not allowed to use *STiles*.

We have decided to implement the sorting step of *STiles* with a GPU radix sort [34] using compute shaders. This sorting strategy performs efficiently enough for our purposes, yielding negligible computation times so the interactivity is not compromised. An implementation of this method in CUDA was also demonstrated to outperform other sorting algorithms in modern GPUs [35]. Another version based on fragment shaders could have been implemented with the aim of enabling older devices to execute *STiles* [36]. However, too many rendering passes are required to carry out the task (with a complexity of $O(n \log^2 n + \log n)$), yielding a serious penalty on mobile GPUs and providing less interactive results.

We have also implemented some other alternatives for progressive ray casting with less satisfactory results. One of our first experiments was based on a naive separation of the high resolution viewport into several tiles. We configured various splitting sizes: we found a grid of $8 \times 8 = 64$ tiles to be the optimal case for this technique, which was raising the completion time to at least one second due to the number of frames (64) needed to finish the rendering. Unfortunately, the transition between the low and high resolution images was not pleasant due to its blocky appearance. This effect could be alleviated by increasing the number of tiles, but this would increase the total rendering time. Furthermore, we implemented and tested an early version of *STiles* that consisted in sorting individual rays instead of tiles, also using compute shaders. Although the idea of sorting seemed sound, the performance also dropped (see Section 5.1). Again, we believe that this is due to the fact that dealing with single rays breaks texture access coherence.

Another approach we implemented was a simple form of progressive ray casting (we name it *Simple* in what follows). It is a screen space refinement method that consists in starting with a low resolution ray casting image (the same used in *STiles*), and then progressively sampling new high resolution rays on the screen surface at each frame until a time budget is expired, finishing when all the pixels of the high resolution image have been computed. The high resolution pixels computed at each frame are accumulated in an extra texture so that they can be reused in subsequent frames. We have tested two different sampling schemes previously used in literature: first, a sampling scheme where the rays are generated randomly (referred to as *Simple random* in the figures), and second, another one where the rays are selected using a regular distribution (labeled as *Simple structured*). The number of refinement steps is variable and depends on the number of rays computed at each frame without exceeding the fixed time budget, which is directly related to the complexity of the rendering process (i.e. resolution of the model, opacity of the transfer function, viewport resolution, etc). The transition effect between the low resolution and the high resolution images was highly smooth, up to the point of almost not noticing the transition. We used the same performance optimizations used in the other methods presented (i.e. ERT and ESS). However, the performance of this approach was worse than our proposed methods (see Section 5.1). Our hypothesis is that the pseudo-random distribution of rays was preventing all kinds of cache usage on the GPU, thus increasing the rendering time at each frame and achieving a much less interactive experience. We present an evaluation of this method in Section 5 together with the evaluation of the proposed techniques.

5. Results

Rendering high quality images of a relatively large dataset on low performance devices such as mobile devices is a task that requires a significant amount of time. We have proposed an incremental approach that splits this process into separated steps that are completed over subsequent frames, so that each step can be executed during an application frame not exceeding an acceptable amount of time. This avoids blocking the application and provides smooth interactivity, allowing the interruption of the high quality rendering at any time if the user desires to continue interacting. Our two proposed methods accomplish this task quickly and in a visually pleasant way. So, from the point of view of the user, the only visible difference is the transition from the low quality image to the high quality image.

We performed several experiments to measure the advantages of both approaches in terms of performance (Section 5.1) and visual quality (Section 5.2). The experiments were run on two mobile devices, a Motorola Nexus 6 (equipped with an Adreno 420 GPU and a screen resolution of 1440×2560) and an Huawei Nexus 6P (equipped with an Adreno 430 GPU and a screen resolution of 1440×2560). On both devices, the viewport resolutions were scaled to half the screen size on both axes for the high resolution ray castings (720×1280, which is still a good resolution due to the small pixel size given on these devices' screens) and to one eight of the screen size for the low resolution ray castings (180×320). We used datasets of different resolutions with transfer functions having different levels of transparency: Vix ($515^2 \times 256$), Head ($512^2 \times 485$), Obelix ($256^2 \times 780$), Chamaleon (512^3) and Melanix ($256^2 \times 602$).

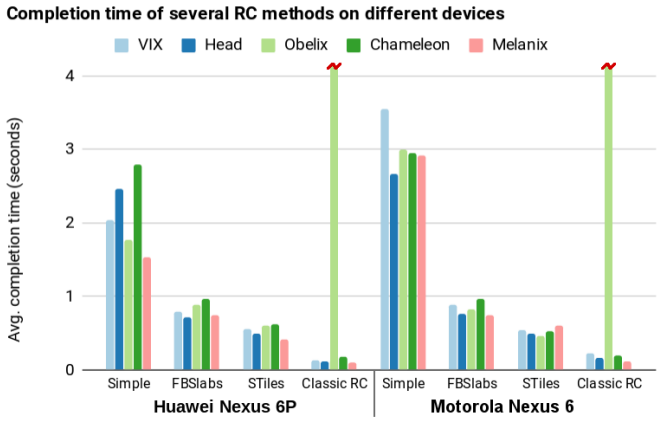

Figure 7: Average completion time (in seconds) of progressive ray casting methods run on different devices for several datasets. These times are an average measure calculated by performing the transition process from 3 different zoom levels with different screen pixel coverages, and 20 different camera positions uniformly distributed on the surface of a surrounding sphere for each zoom level (60 camera configurations in total). Note that the measurements for the *Classic RC* complete faster in average, which is the expected behaviour as the rendering task is not split over several frames. However, this does not implies better interactivity than progressive methods, because these distribute the workload over several frames, returning the control to the application more frequently. Note also the high bars in the *Classic RC*, indicating that some renderings were not completed due to an application crash.

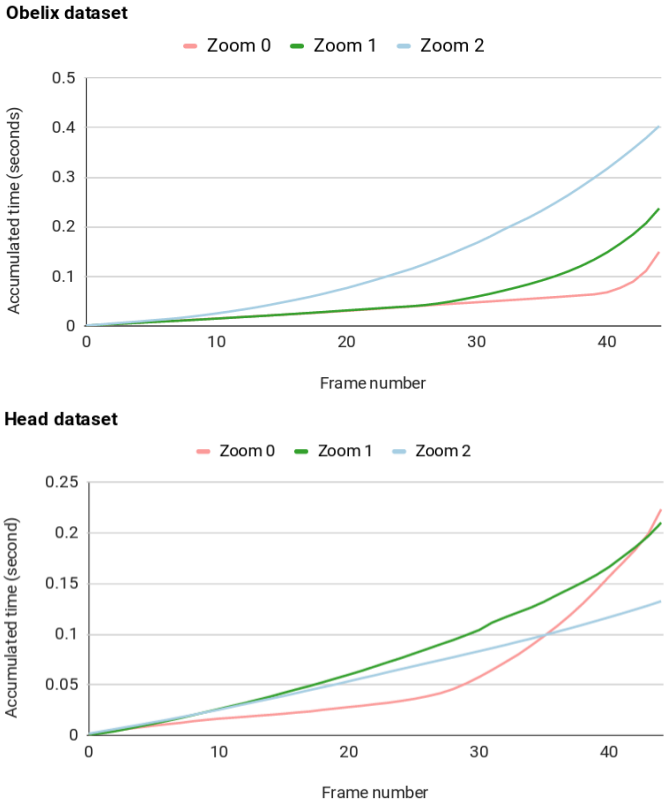

Figure 8: Charts showing the accumulated time over subsequent frames in STiles for two datasets. The lines correspond to different levels of camera zoom, corresponding Zoom 0 to the smallest, and Zoom 2 to the largest screen pixel coverage. A fixed number of tiles is rendered at each frame, and the tiles have been previously sorted by increasing cost. Last frames obviously take longer to finish.

5.1. Transition from Low-Res to High-Res: Performance

FBSlabs distributes the workload over time by splitting the rays into segments. At each frame of this progressive method, a limited number of ray casting samples is fixed, so the maximum number of samples within the ray casting shader, for a single frame, is $O(V_w \times V_h \times N)$, where $V_w \times V_h$ is the total number of pixels in the viewport and N is the fixed number of samples to take from each ray segment during a single frame of the incremental rendering. We have fixed $N = 40$ in our experiments so that a small loop is performed for each pixel in the viewport at each frame. Besides the rendering of each slab, the amount of time required for blending both, the low resolution and the high resolution images, is negligible. Some results are shown in Figure 7 (*FBSlabs* series). In average, our experiments obtain completion times under 1 second for the tested models.

We have tried to improve *FBSlabs* in order to store per-ray accumulated opacity after each frame so that a global ERT is enabled. However, this implemetation requires an extra pass to copy the high resolution results into another texture that can be queried during the next frame to know whether or not the current ray/pixel was completed and can be discarded. Unfortunately, this extra pass incurs a penalty that incurs in a time penalty that is larger than the benefits obtained from ERT. The algorithm can still perform per-slab early ray termination, but it will not avoid starting the ray traversal for the next slab in the next frame.

In *STiles*, the workload is split into screen-space tiles that can have different costs depending on the length of the rays they contain, and sorted before proceeding to the progressive ray casting step. The sorting step cost is actually negligible and it is computed only once after each user interaction (see step 2 of Figure 5). We base our strategy on the experimental results shown in Figure 8. If tiles are sorted by increasing cost, it can be observed that the accumulated time of the incremental rendering along subsequent frames (when a fixed number of tiles is rendered at each frame) increases in a non-linear way, due to the obvious fact that rendering the first tiles is faster than rendering the last ones. Our strategy, based on the charts in Figure 8, is to render more tiles in the first frames and a lower number of tiles in the last frames to compensate for their higher cost. Based on the shape of the curves in these charts, we estimate a tile budget for each subsequent frame that guarantees an estimated time budget of 0.1 seconds per frame. Estimated tile budgets are decreasing from the first frame to the last one, resulting on a greater number of tiles being rendered in the first frames and on a stable frame rendering time. As shown in Figure 7, we achieve completion times faster than *FBSlabs* method (approximately half the time for all the tested datasets on all devices).

One could argue that rendering tiles in ascending order in *STiles* implies rendering big empty regions of the screen first (which should have cost zero) whenever the footprint of the proxy geometry is much smaller than the actual screen resolution. The ideal procedure would be to directly discard those tiles without effective work to process, or those not overlapping the proxy geometry. However, discarding tiles with zero cost

	Nexus 6											
	Simple			FBSlabs			STiles			Classic RC		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Vix	8.03635	11.6122	8.84522	10.7524	32.6419	20.6254	12.5105	23.0776	16.5257	2.24384	11.2076	4.97987
Head	8.16621	11.1880	9.73117	17.5999	37.4925	25.4241	11.0722	23.2729	16.2087	3.30920	11.4138	6.44491
Obelix	7.85773	10.2932	8.83859	13.8662	40.3193	23.9071	9.29148	26.8632	17.0683	×	13.3783	6.71495
Chamaleon	7.90516	10.8366	9.75879	17.1355	37.2815	25.3579	13.5114	23.7053	16.8985	2.04548	6.94170	4.48132
Melanix	7.92802	10.2918	8.65489	15.1961	42.8415	25.5537	13.0587	28.4416	19.7347	4.16008	19.5485	9.98187
	Nexus 6P											
	Simple			FBSlabs			STiles			Classic RC		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Vix	9.04296	10.8118	9.65829	18.3297	47.705	28.8664	13.4198	24.1418	16.6214	3.84157	16.7853	7.96326
Head	7.8058	10.5197	9.65324	23.0565	51.6828	31.5486	13.7861	24.4392	17.263	4.03957	14.8991	8.70402
Obelix	8.65025	12.5418	9.68395	17.7137	48.3289	26.762	9.68402	26.2899	16.4421	×	19.3723	9.07656
Chamaleon	8.82493	12.9486	9.59793	17.1363	47.9752	26.1988	12.1175	21.7779	15.7702	2.34358	11.5424	6.37945
Melanix	9.18801	13.3794	10.1308	18.5456	54.088	26.9182	13.1736	32.586	20.3036	4.99895	22.5877	12.6974

Table 1: These frame rates reflect the interactivity of the presented progressive ray casting methods with respect to a classic non-progressive ray casting algorithm on two different mobile devices. All progressive methods perform interactively in all cases during the generation of the high resolution image, being *FBSlabs* the more interactive, followed by *STiles* and being the *Simple* progressive method in third place. Note, however, that the frame rates provided by a classic non-progressive ray casting provides worse frame rates and hence bad interactivity in average, and provoking occasional application crashes as shown in Figure 7.

is not reliable, as tile costs are computed from a low-resolution image rendering, and furthermore, we actually classify each tile by the cost queried from a single sample position at its center. However, this issue is not a problem, as the rendering of empty, and almost empty tiles, completes instantly when the fragment shader discards rays not intersecting the proxy geometry, so it is actually normal completing all the empty regions and part of the effective ray casting workload during the first frame.

As previously commented, we tested an initial version of *STiles* that consisted in sorting individual rays, achieving poorer performance. We were then inspired by an analysis of the rasterization patterns followed by several GPUs in [37], where the authors were able to reveal the order in which pixels are rendered by the GPUs by means of using atomic counters in a fragment shader. Based on this observation we performed an analysis of the performance by running some tests, packing groups of rays in tiles of several sizes (see Figure 9). As expected, increasing the tile size boosts performance. The rationale behind this is that packing neighbouring rays together takes advantage of the 3D texture cache. Following this argumentation, performing the whole rendering at once would achieve an optimal result. However, the measurements shown in Figure 9 are averaged over a big variety of camera configurations where some renderings are generated very quickly and others can take much longer (e.g. the Body model seen from above through its longest axis), and they could provoke the aforementioned application crash issue if not split over time. We finally decided to use a minimum tile size of 8×8 pixels, as the performance gain considerably decreases for larger tile sizes. As shown in Section 5.2, this tile size achieves a good compromise between the rendering time and the perceived transition between different frames.

We also tested the performance of the *Simple* progressive ray caster. The achieved completion times were the higher among all methods (see Figure 7). This is due to the distribution pattern followed to generate rays for the high resolution ray casting. It does not take into consideration any locallity pattern, breaking the spatial coherence and not making possible the use of the 3D texture cache, finally increasing the total completion time. In addition, we executed performance tests of a classic non-progressive ray casting algorithm to compare the achieved times with the result of our proposed progressive methods. The average rendering times obtained may seem lower than our two proposals (see Figure 7, *Classic RC*). However, these are averaged numbers only from successful frames. Other images, taking longer to be rendered stall the application until finishing, not giving the user the opportunity to interact. Some others cannot even be averaged as they make the application crash due to long stalls (this is the case of the *Obelix* dataset when visualized along its longest axis, as the used transfer function is barely opaque, and that generates very long rays). Furthermore, it is desirable to receive partial results of the final image right after finishing interacting (even if it takes a bit longer to complete the image), which gives the user a hint to perceive that the application is actually working. This performance is again not offered by classic non-progressive ray casting algorithms.

Some extra tests were performed in order to measure and compare the interactivity of the presented progressive ray casting methods. As seen in Table 1, all progressive methods present an acceptable frame rate in all cases during the generation of the high resolution image, being *FBSlabs* the more interactive, followed by *STiles*, and being the *Simple* progressive method in third place. Note, however, that the classic non-progressive ray casting provides worse frame rates and hence bad interactivity in average, and provokes application crashes occasionally, as shown in Figure 7.

5.2. Transition from Low-Res to High-Res: Visual Effect

The visual effect of the transition between low resolution and high resolution images obtained by *FBSlabs* and *STiles* is quite different. Figures 4, 6, 14, 15, 16 and 17 show the progression of each method during the transition time with renderings of several datasets, visualized with transfer functions designed with different colors and opacities. The accompanying video depicts the progression effect over time better.

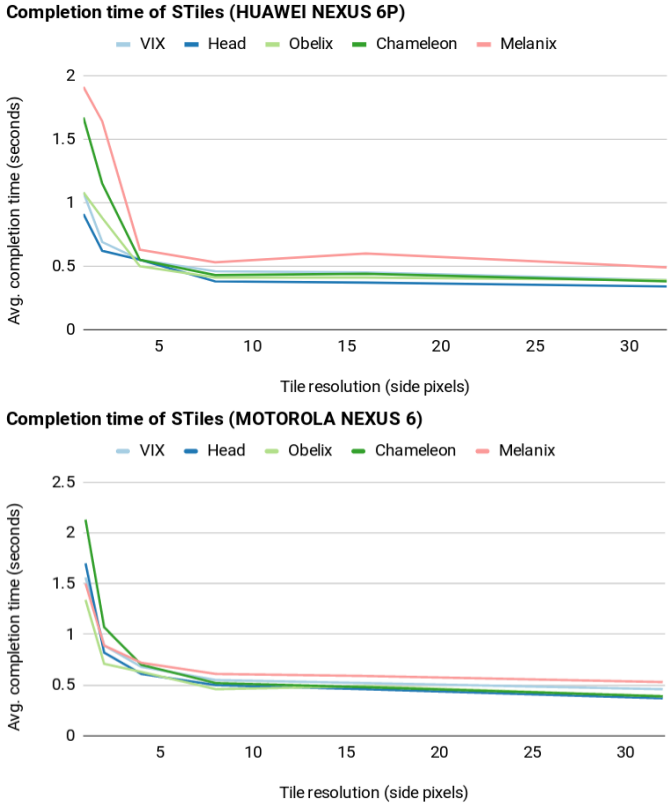

Figure 9: These charts show the overall completion times (in seconds) obtained for the *STiles* algorithm under several tile size configurations. The tests were run on two different devices (Huawei Nexus 6P on the top, Motorola Nexus 6 at the bottom) with several datasets. The tested tile sizes were: 1^2, 2^2, 4^2, 8^2, 16^2 and 32^2. We can see how the completion time decreases as the tile size increases. More precisely, the performance gain is particularly low for sizes greater than 8^2, which is actually the size of the rasterization patterns used by those GPUs.

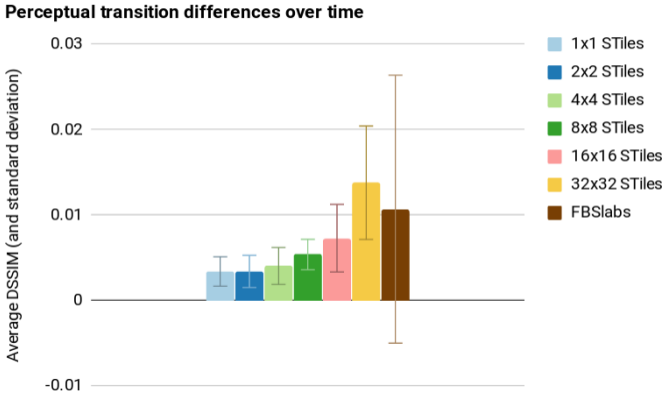

Figure 10: This chart shows the measured average perceptual error (and its standard deviation) on the transition process (going from the low-resolution image to the high-resolution image). The perceptual metric used is the structural dissimilarity metric (DSSIM). The average error was computed using pairs of consecutive frames in several series of the incremental ray casting process. We can observe that the transition becomes perceptually more evident (i.e. has a higher error measure) as the tile size increases, being significantly greater for tile sizes greater than 8^2 (note that 16^2 has a considerably higher standard deviation).

The progressive *FBSlabs* method has the effect of the high resolution image appearing on top of the low resolution one (see Figure 11, *FBSlabs*) and completes gradually replacing the low resolution image in front-to-back order. During the incremental rendering, the final color that is presented onto the screen is the composition of the high resolution image on top of the remaining part of low resolution image using alpha blending. An issue regarding this way of compositing images is that we are mixing viewport resolutions. In the context of ray casting, this means two things. The first one is the fact that the rays in the low-resolution image do not perfectly match rays in the high-resolution image. And the second one is that we are performing an upsampling of the low-resolution image, so we are interpolating color to match the sizes of both images. This sometimes results in slight seam artifacts revealed in the boundary between the high resolution and the low resolution models.

STiles also reveals the final high quality image gradually, but in this case, small tiles with the corresponding part of the high resolution image appear in a pseudo-random order (see Figure 11, *STiles*). It also gives the impression of completing the result in some sort of front-to-back order (or back-to-front order, it actually depends on the sorting strategy) but each tile with high resolution color that has been computed completely replaces the initial low resolution color, instead of compositing the high resolution color over the low resolution color as in *FBSlabs* (see the accompanying video to appreciate the effect over time). We can choose between sorting tiles in increasing or decreasing order of ray cost. In the first case, tiles with small cost (e.g. those with rays that become completely opaque very quickly) are rendered first. This way, models visualized with transfer functions designed to reveal opaque isosurfaces exhibit a transition effect that gives the perception of most parts of the final image appearing first and then the silhouettes appearing in the end. A reverse sorting strategy, starting from tiles with an estimated high cost and then rendering tiles in decreasing order gives the contrary visual effect: first, translucent areas and most silhouettes are revealed, and then opaque areas with little translucent component are computed in last place. We decided to sort tiles by increasing order because, in most cases, the effect it achieves is more desirable, and furthermore, the transition achieved gives the perception of completing sooner due to the fact of rendering more tiles in the first frames.

As explained in Section 5.1, we empirically determined a lower boundary of the tile size (in pixels) based on an analysis of the GPU rasterization pattern [37] and a series of experiments regarding performance (Figure 9). These experiments show a tendency to gain performance when increasing the tile size. However, *STiles* performs a tile-based rendering, and it consequently presents a blocky transition effect that becomes more evident when the tiles are too large. To determine an appropriate tile size, we also performed a series of experiments to measure the transition changes over time using a perceptual structural dissimilarity metric (DSSIM). Figure 10 shows averaged perceptual differences over time. The perceptual differences shown in the chart are obtained by comparing each intermediate frame with the previous one. Based on the obtained results, we decided to fix the tile size to 8×8, as the perceptual

Figure 11: Detail of an intermediate step during the high resolution transition process (Head dataset $512^2 \times 485$ high res., $128^2 \times 122$ low res.). In *FBSlabs*, the transition boundary is more evident and reveals patterns generated by the fact that ray sampling proceeds front-to-back from the proxy geometry. The boundary is less perceivable in *STiles*, which furthermore has a pseudo-random transition pattern that makes it less evident over time.

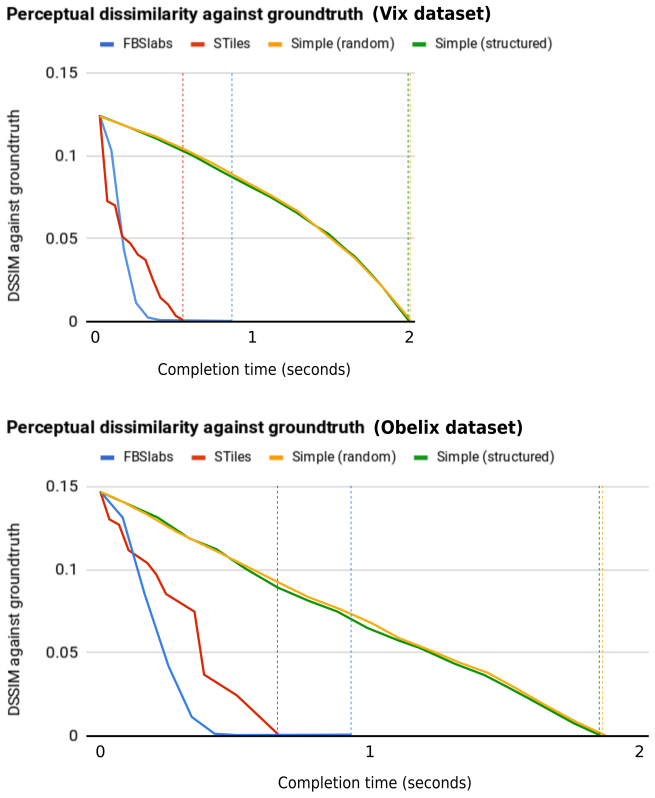

Figure 12: Perceptual changes of progressive methods over time. Vertical colored lines indicate the completion time in each case. The data was taken from rendering the *Vix* and the *Obelix* datasets (see Figures 6 and 16) and comparing each frame with the final image (ground truth). It can be observed that *STiles* is the fastest method and has a smooth convergence to the ground truth.

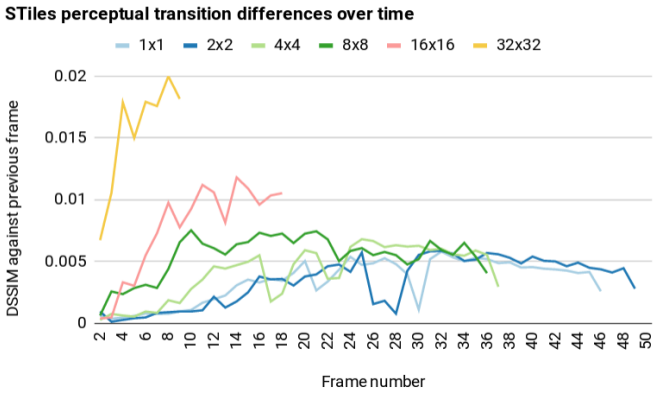

Figure 13: Perceptual changes of *STiles* over time using different tile sizes. Data taken from rendering the *Vix* dataset (see Figures 4 and 6) and comparing each pair of subsequent frames in the timeline. Larger tile sizes achieve higher perceptual changes between subsequent frames. This is actually normal considering that the final image completion is usually achieved in less frames when using larger tile sizes. Note that tiles of size 8×8 and smaller achieve similar measures over time, yet tiles of size 8×8 take less frames to finish among those *small* tile sizes (and less time, see Figures 9 and 7).

differences increase for larger tile sizes. This size is actually the lower boundary determined in the previous section, and also the size of the tiles generated by the rasterization process on these GPUs. This size is small enough so that the blocky nature of this method is not evident or annoying during the transition between the low resolution and the high resolution images.

We did another set of tests to measure and evaluate the quality of transition on several datasets, also using perceptual metrics (DSSIM). Figure 12 shows the perceptual transition profile of the *FBSlabs* and *STiles* progressive methods, and of the two different approaches of the *Simple* progressive ray caster, one distributing rays in a pseudo-random order (random), and another in a more structured way (structured). These tests were done using the *Vix* and the *Obelix* datasets (see Figures 6 and 16), which are visualized using transfer functions with different levels of transparency. The charts show the perceptual image variation of each frame with respect to the final (ground truth) image. The vertical lines indicate the completion time of each method. In both charts, we can see how *STiles* is the fastest method, and its more uniform convergence to zero indicates that it produces a more smooth transition. We can also observe how *FBSlabs* shows a less uniform slope in its overall time interval in the charts, and after approaching the ground truth, it keeps on executing during several frames until the whole model has been rendered. This quick convergence is due to the front-to-back nature of the method, as the front part of the model usually covers most part of the image, yet the back part of it has smaller visual impact on the final result. This results in a sudden change in the first frames and very subtle variations in the last ones. The *Simple* random and *Simple* structured techniques, like *STiles*, also have a smooth and constant visual transition effect, but their total completion times are longer. Summarizing, these observations confirm the perception we had when analyzing the running application and our preference towards *STiles*, as it quickly converges to the final image and keeps a gradual and smooth transition over time.

Figure 13 shows DSSIM measurements of each frame of the progressive rendering with respect to its previous frame for different tile size configurations in *STiles*. In this case, the charts show that the biggest tile sizes achieve a higher error, meaning that the transition is less smooth and more perceivable. However, tiles of size 8×8 and smaller have a similar profile. Taking this into account and considering the performance results in the previous section (see Figure 9), we decided to use tiles of size 8×8 as the default option.

5.3. Discussion

Both *FBSlabs* and *STiles* are usable when generating progressive renderings of volume data. The presented performance tests show that they enable less powerful devices to render big volumes of data otherwise not feasible. Table 2 summarizes the main features of the two proposed algorithms. We recommend using *STiles* over *FBSlabs* whenever possible. It fits devices with OpenGL ES 3.1 (needed for the compute shaders). The results obtained for *STiles* are better both in performance and in visual quality as demonstrated in the previous sections. It completes the high quality image in less time than *FBSlabs* and the perceptual variation over time as the transition advances is smaller, a fact that matches our visual assessment (see the accompanying video). Not far from it, however, *FBSlabs* is a good candidate to use in less powerful devices that do not provide compute shaders (only available from OpenGL ES 3.1). Furthermore, even when running on more capable hardware, *FBSlabs* is a good choice on platforms such as WebGL, whose standard still does not support modern features such as compute shaders. Moreover it could even be adapted for older devices that do not provide 3D textures using a scheme based on flat 3D textures or stacked 2D textures, for instance.

6. Conclusions and Future Work

In this paper, we have proposed a multiresolution architecture based on ray casting aimed at achieving the interactive rendering of volume ray casting in less powerful devices, such as mobile phones and PCs with low-end and old graphics chips. We use a low resolution dataset to perform interactive visualizations during user interaction, and the higher resolution version of the same dataset (that still fits the target's GPU memory) to perform a high quality visualization each time the user stops interacting. We use a set of techniques such as a feature-preserving downsampling filter and adaptive transfer functions in order to improve the quality of coarse resolution datasets.

Our main contributions are two scalable methods for the progressive ray casting of high resolution datasets that are able to decouple the rendering process into separated batches that can be rendered over subsequent frames: *FBSlabs* and *STiles*. These algorithms are able to provide an interactive user experience without application stalls at any time. Based on the performed experiments, we conclude that *STiles* achieves better results in both performance and visual quality than *FBSlabs*, as presented in Section 5. *FBSlabs* is, however, a good candidate for less up to date devices that do not provide modern GPU features (e.g. compute shaders).

Regarding *STiles* a slight improvement would be the ability to split the current individual ray batches into several parts. It is not likely that our algorithms are going to deal with volume datasets large enough to make the device stall by only rendering a single ray group. However, that could happen if rays were long enough, which could be solved by also allowing incremental rendering of individual tiles.

Current sizes of really large datasets ($\geq 1024^3$) cannot fit current GPUs' memory specifications. A possible way to extend our architecture is the implementation of an out-of-core block based scheme that allows fetching blocks as needed during the high resolution rendering process, so our progressive rendering algorithm could require the needed blocks from the storage memory or server at each frame. At first sight it seems that the implementation of a block-based on-demand architecture like this could be easier to extend *FBSlabs*, which already performs an object space partition to carry out the progressive rendering, rather than *STiles*, which is a screen space approach.

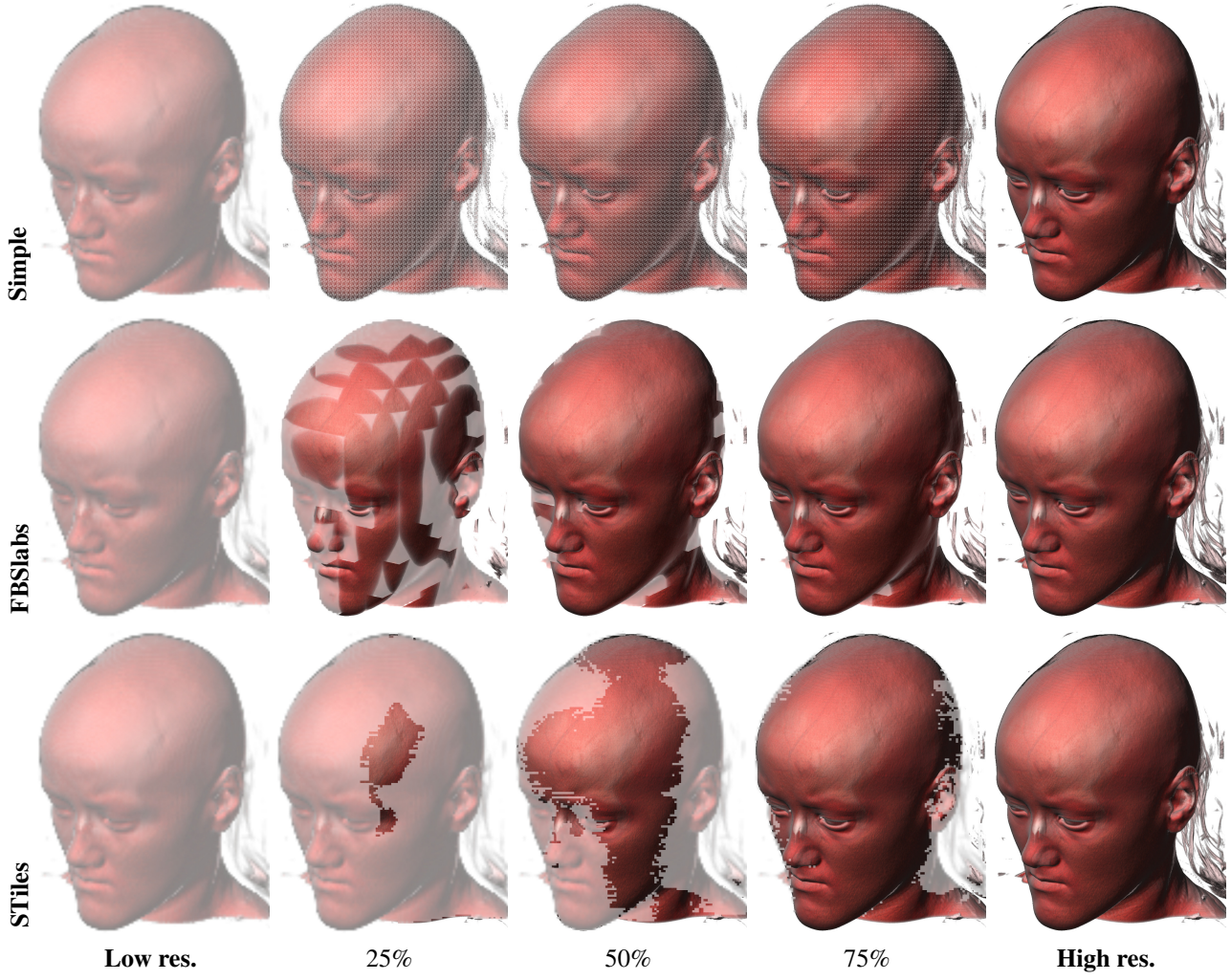

Figure 14: Illustration of the transition in the presented algorithms for the Head dataset ($512^2 \times 485$ high res., $128^2 \times 122$ low res.). These figures do not correspond to the actual rendering, but we modified them in order to show which parts of the image are updated over subsequent frames in both algorithms: the region that has not yet been updated with the high quality rendering is shown with a semi-transparent look. Note that *Simple* has the more incremental transition. Note also that *FBSlabs* has homogeneous boundaries that are easier to perceive during the progression than *STiles*, and *STiles* provides a pseudo-random transition pattern that is more difficult to notice during the incremental rendering (see Figure 11).

Figure 15: Melanix dataset ($256^2 \times 602$ high res., $64^2 \times 151$ low res.). Transition effect of the two proposed incremental ray casting algorithms using a transfer function with almost opaque colors.

	FBSlabs	STiles
OpenGL version	Requires OpenGL ES 3.0 or lower if the 3D volume is managed with 2D textures.	Requires OpenGL ES 3.1 because it needs compute shaders.
Transition effect	High-resolution image appearing front to back. Major changes occur during the first frames. More perceivable seams between low-resolution and high-resolution models.	Better DSSIM perceptual results. Transition occurs more regularly distributed over time. Pseudo-random substitution pattern of the low-res image by the high-res one.
Transition time	Good average completion times. A small number of ray casting samples is fixed at each frame. High interactivity rate.	Better average completion times. A time budget is fixed for each frame that cannot be exceeded. At each step, as many tiles as possible are rendered. Good interactivity rate.

Table 2: Characteristic features of *FBSlabs* and *STiles* methods for progressive ray casting.

7. Acknowledgements

The authors wish to thank the anonymous reviewers. Their valuable comments and suggestions helped improving the paper. The material in this paper is based upon work supported by the Spanish *Ministerio de Economia y Competitividad* and by FEDER (EU) funds under the Grants No. TIN2014-52211-C2-1-R and TIN2017-88515-C2-1-R.

References

[1] Fogal T, Krüger J. Tuvok, an Architecture for Large Scale Volume Rendering. In: Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization. 2010,URL: http://www.sci.utah.edu/~tfogal/academic/tuvok/Fogal-Tuvok.pdf.

[2] Noguera J, Jiménez J. Visualization of very large 3d volumes on mobile devices and webgl. In: 20th WSCG international conference on computer graphics, visualization and computer vision. WSCG. Citeseer; 2012,.

[3] Noon CJ. A Volume Rendering Engine for Desktops, Laptops, Mobile Devices and Immersive Virtual Reality Systems using GPU-Based Volume Raycasting. Master's thesis; Iowa State University; 2012.

[4] Mobeen MM, Feng L. Ubiquitous medical volume rendering on mobile devices. In: International Conference on Information Society (i-Society 2012). 2012, p. 93–8.

[5] Movania MM, Chiew WM, Lin F. On-site volume rendering with GPU-enabled devices. Wirel Pers Commun 2014;76(4):795–812.

[6] Hadwiger M, Kniss JM, Rezk-salama C, Weiskopf D, Engel K. Real-Time Volume Graphics. Natick, MA, USA: A. K. Peters, Ltd.; 2006. ISBN 1568812663.

[7] Balsa Rodríguez M, Gobbetti E, Iglesias Guitián JA, Makhinya M, Marton F, Pajarola R, et al. State-of-the-Art in Compressed GPU-Based Direct Volume Rendering. Computer Graphics Forum 2014;33(6):77–100.

[8] Beyer J, Hadwiger M, Pfister H. A Survey of GPU-Based Large-Scale Volume Visualization. Proceedings EuroVis 2014 2014;.

[9] Crassin C, Neyret F, Lefebvre S, Eisemann E. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games. I3D '09; New York, NY, USA: ACM; 2009, p. 15–22.

[10] Fogal T, Schiewe A, Krüger J. An analysis of scalable GPU-based ray-guided volume rendering. In: Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on. IEEE; 2013, p. 43–51.

[11] Callahan SP, Silva CT. Accelerating unstructured volume rendering with joint bilateral upsampling. J Graphics, GPU, & Game Tools 2009;14(1):1–15. URL: https://doi.org/10.1080/2151237X.2009.10129271.

[12] Levoy M. Volume rendering by adaptive refinement. The Visual Computer 1990;6(1):2–7. URL: http://dx.doi.org/10.1007/BF01902624.

[13] Kratz A, Reininghaus J, Hadwiger M, Hotz I. Adaptive screen-space sampling for volume ray-casting. Tech. Rep. 11-04; ZIB; Takustr.7, 14195 Berlin; 2011.

[14] Frey S, Sadlo F, Ma KL, Ertl T. Interactive progressive visualization with space-time error control. IEEE Transactions on Visualization and Computer Graphics 2014;20(12):2397–406.

[15] Schiewe A, Anstoots M, Krüger J. State of the Art in Mobile Volume Rendering on iOS Devices. In: Bertini E, Kennedy J, Puppo E, editors. Eurographics Conference on Visualization (EuroVis) - Short Papers. The Eurographics Association; 2015,.

[16] Noguera JM, Jiménez JR. Mobile volume rendering: Past, present and future. IEEE transactions on visualization and computer graphics 2016;22(2):1164–78.

[17] Lamberti F, Sanna A. A solution for displaying medical data models on mobile devices. SEPADS 2005;5:1–7.

[18] Hachaj T. Real time exploration and management of large medical volumetric datasets on small mobile devices—evaluation of remote volume rendering approach. International Journal of Information Management 2014;34(3):336–43.

[19] Gutenko I, Petkov K, Papadopoulos C, Zhao X, Park JH, Kaufman A, et al. Remote volume rendering pipeline for mHealth applications. vol. 9039. 2014, p. 903904–.

[20] Campoalegre L, Brunet P, Navazo I. Interactive visualization of medical volume models in mobile devices. Personal and Ubiquitous Computing 2013;17(7):1503–14.

[21] Congote J, Segura A, Kabongo L, Moreno A, Posada J, Ruiz O. Interactive visualization of volumetric data with webgl in real-time. In: Proceedings of the 16th International Conference on 3D Web Technology. ACM; 2011, p. 137–46.

[22] Mobeen MM, Feng L. High-performance volume rendering on the ubiquitous webgl platform. In: High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on. IEEE; 2012, p. 381–8.

[23] Movania MM, Lin F. Real-time volumetric lighting for webgl. WebGL Insights 2015;:261.

[24] Schultz C, Bailey M. Interacting with large 3d datasets on a mobile device. IEEE Computer Graphics and Applications 2016;36(5):19–23.

[25] Noguera JM, Jiménez JR, Ogáyar CJ, Segura RJ. Volume rendering strategies on mobile devices. In: GRAPP/IVAPP. 2012, p. 447–52.

[26] Noguera JM, Jiménez JJ, Osuna-Pérez MC. Development and evaluation of a 3d mobile application for learning manual therapy in the physiotherapy laboratory. Computers & Education 2013;69:96–108.

[27] Mobeen MM, Feng L. Mobile visualization of biomedical volume datasets. J Internet Technol Secur Trans 2012;1(2):52–60.

[28] Balsa Rodríguez M, Alcocer PPV. Practical volume rendering in mobile devices. In: International Symposium on Visual Computing. Springer; 2012, p. 708–18.

[29] Xin Y, Wong HC. Intuitive volume rendering on mobile devices. In: 2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI). 2016, p. 696–701.

[30] 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; New York, NY, USA: ACM. ISBN 1-58113-407-X; 2001, p. 9–16.

[31] Díaz-García J, Brunet P, Navazo I, Pérez F, Vázquez P. Feature-

preserving downsampling for medical images. In: EuroVis, posters. 2015,.
[32] Díaz-García J, Brunet P, Navazo I, Perez F, Vázquez PP. Adaptive transfer functions. Vis Comput 2016;32(6-8):835–45.
[33] Krüger J, Westermann R. Acceleration Techniques for GPU-based Volume Rendering. In: Proceedings of the 14th IEEE Visualization 2003 (VIS'03). VIS '03; Washington, DC, USA: IEEE Computer Society. ISBN 0-7695-2030-8; 2003, p. 38–.
[34] Harada T, Howes L. Introduction to GPU Radix Sort. 2011.
[35] Satish N, Harris M, Garland M. Designing efficient sorting algorithms for manycore gpus. In: 2009 IEEE International Symposium on Parallel Distributed Processing. 2009, p. 1–10.
[36] Kipfer P, Westermann R. Improved gpu sorting. In: Pharr M, editor. GPU Gems 2. Addison-Wesley; 2005, p. 733–46.
[37] JeGX . OpenGL 4.2 Atomic Counters: Rasterization Pattern, Helper for Rendering Optimization (Windows, Linux). http://www.geeks3d.com/20131031/opengl-4-2-atomic-counters-rasterization-pattern-helper-for-rendering-optimization-windows-linux/; 2013. [Online; accessed 14-December-2016].

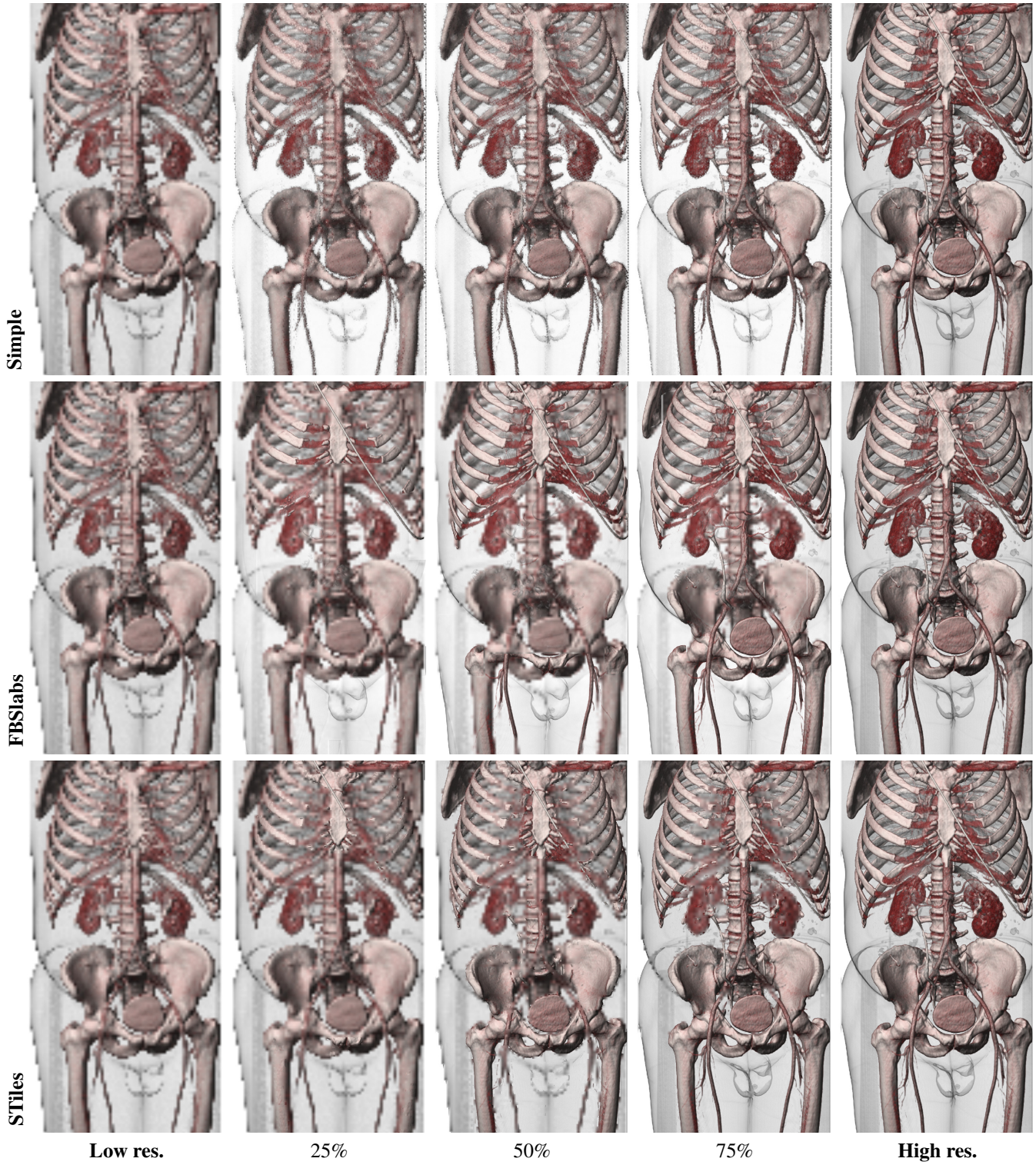

Figure 16: Obelix dataset ($256^2 \times 780$ high res., $64^2 \times 195$ low res.). Transition effect of the *Simple*, *FBSlabs* and *STiles* incremental ray casting algorithms using a transfer function with some opaque colors (bones, kidneys, etc) and semitransparent colors (skin).

Figure 17: Chameleon dataset (512^3 high res., 128^3 low res.). Transition effect of the two proposed incremental ray casting algorithms using a transfer function with some opaque colors (bones, muscles, etc) and semitransparent colors (skin). Although we are mainly focusing on medical datasets, the presented algorithms are perfectly suited for any other kinds of volume datasets such as this one.