GPU Ray Casting

Ricardo Marques
Dep. Informática, Universidade do Minho
VICOMTech
ricjmarques@gmail.com

Luís Paulo Santos
Dep. Informática, Universidade do Minho
Braga, Portugal
psantos@di.uminho.pt

Peter Leškovský   Céline Paloc
VICOMTech
Donostia - San Sebastian, Spain
{pleskovsky,cpaloc}@vicomtech.org

Abstract

For many applications, such as walk-throughs or terrain visualization, drawing geometric primitives is the most efficient and effective way to represent the data. In contrast, other applications require the visualization of data that is inherently volumetric. For example, in biomedical imaging, it might be necessary to visualize 3D datasets obtained from CT or MRI scanners as a meaningful 2D image, in a process called volume rendering. As a result of the popularity and usefulness of volume data, a broad class of volume rendering techniques has emerged. Ray casting is one of these techniques. It allows for high quality volume rendering, but is a computationally expensive technique which, with current technology, lacks interactivity when visualizing large datasets, if processed on the CPU. The advent of efficient GPUs, available on almost every modern workstations, combined with their high degree of programmability opens up a wide field of new applications for the graphics cards. Ray casting is among these applications, exhibiting an intrinsic parallelism, in the form of completely independent light rays, which allows to take advantage of the massively parallel architecture of the GPU. This paper describes the implementation and analysis of a set of shaders which allow interactive volume rendering on the GPU by resorting to ray casting techniques.

Keywords

Volume rendering, GPU ray casting, volume visualization

1. INTRODUCTION

Three dimensional volume datasets are frequently used in the scientific community. They are obtained by simulation, sampling or modeling. In economics or fluid dynamics, for example, numerical simulations can generate large scale volumetric datasets. In medical imaging, different scanning techniques such as Magnetic Resonance Imaging (MRI) or Computed Tomography (CT) are used to collect samples of the interior of the human body, which are stored as 3D datasets [Johnson 04].

These datasets may be visualized in three dimensions, in order to allow specialists to interpret the information. In traditional computer graphics, 3D objects are created using surface representations, by drawing geometric primitives that create polygonal meshes [Levoy 88]. However, using surface rendering techniques to display volumetric data results in the loss of a dimension of information [Johnson 04, Hadwiger 06]. For example, in CT datasets, the useful information is not only contained on extracted iso-surfaces, but within the iso-surfaces as well. Therefore, several volume rendering techniques were developed to visualize the entire 3D data as a single 2D image. Volume rendering techniques convey more information than surface rendering methods, but at the cost of increased algorithm complexity and, consequently, increased rendering times [Bruckner 08].

Ray casting [Kajiya 84] is one of these techniques. It evaluates the color of each pixel in the final image by shooting a ray through the scene starting from the viewer position. If the ray hits the volume, the color of the pixel is calculated by sampling the data values along the ray at a finite number of positions in the volume and combining them together. This technique, however, has a limitation when executed on CPUs: for large volume datasets and close viewing planes that maximize the number of rays which must be shot, the time to render a single image is too high to allow for a real time visualization.

Driven by the demand of the game industry, the performance of modern GPUs has exceeded the computational power of CPUs both in raw numbers and in the growth rate [Stegmaier 05, Weiskopf 07]. Therefore, GPUs appear as an interesting opportunity to execute heavy algorithms, for which the CPUs cannot give a real time response. Modern graphics cards are characterized by the following features [Kruger 03, Scharsach 05]:

- A massively parallel architecture
- A separation into two distinct units (vertex and fragment shader) that can double performance if workload can be split
- Fast memory and memory interface
- Dedicated instructions for graphical tasks
- Vector operations on 4 floats that are as fast as scalar operations
- Trilinear interpolation is automatically (and extremely fast) implemented in the 3D texture

The ray casting algorithm fits modern GPUs. It exhibits an intrinsic parallelism, in the form of completely independent light rays, which allows to take advantage of the massively parallel architecture of the GPU. But as a new technology, GPU ray casting is not well established yet. Appropriate libraries implementing the technique which are compatible with graphics processors from the two main manufacturers, NVidia and ATI, cannot be found.

The work presented in this paper is motivated by the creation of a Human Atlas visualization tool which allows the user to navigate through human medical datasets (obtained from CT or MRI scanners) in real time, using direct volume rendering. Although no original techniques are described, the paper presents a detailed overview of the current state of the art in GPU ray casting for volume visualization and a thorough comparison of different compositing and antialising techniques.

2. RELATED WORK

In the field of GPU-Based volume rendering [Weiskopf 07, Hadwiger 06], there are two distinct approaches to render 3D datasets at interactive rates: a texture based approach, and a ray casting based approach. Below, these two approaches are presented, along with their main features, advantages and drawbacks.

2.1. Texture Based Approach

The texture based approach was the first GPU-Based volume rendering technique, originally presented by Cullip and Newman [Cullip 94], and further developed by Cabral et al. [Cabral 94]. In this approach, the volumetric data is stored in the GPU memory as a stack of object aligned 2D textures [Westermann 98] or as a viewport aligned 3D texture [RS00]. These textures are then mapped onto a sequence of semi-transparent 2D slices, called proxy geometry, using the built in hardware texture interpolation. The function of the proxy geometry is to provide a sequence of polygons where the texture slices will be displayed. Finally, the proxy geometry is rendered in back to front order, exploiting the hardware per fragment operations and alpha blending capabilities of the GPU.

The 2D texture approach requires three copies of the volume dataset, each of them aligned with one of the main axis of the object (see Figure 1).

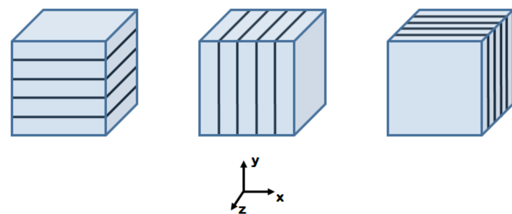


Figure 1: Object aligned sampling surfaces.

This increases three times the amount of memory needed to store the volumetric data. During the visualization process, one of the stacks is chosen to be displayed, depending on the current viewing direction. The chosen stack is the one corresponding to the axis most parallel to the viewing vector. To map the texture slices to the proxy geometry, 2D interpolation within each slice of the stack is used. When a given texture slice is mapped, the information from the previous and the next slices is not taken into account. This causes the final result to have lower quality, usually arising in visible artifacts [Hadwiger 06]. The way to increase the quality of the final image, thus removing these imperfections, is to increase the sampling rate by incrementing the number of texture slices that store the volume on the graphics card, causing the amount of memory necessary to be even larger.

Compared with the 2D texture solution, the 3D texture based approach is superior, removing some of the significant drawbacks while preserving almost all the benefits [Hadwiger 06]. With 3D textures only one copy of the volume dataset is necessary, because trilinear interpolation allows for the extraction of slices in arbitrary directions, for example diagonally (see Figure 2).

This reduces the size of the volume in the graphics card memory when compared to the 2D texture approach. The volume is sliced by a set of planes parallel to the viewing direction, and the resulting slices are composed to achieve the final image. Furthermore, the use of trilinear interpolation instead of bilinear re-

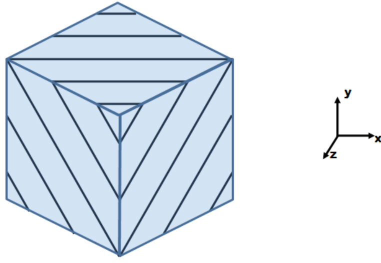


Figure 2: Viewport aligned sampling surface.

sults in higher image quality. It also provides a natural way to increase the sampling rate only by increasing the number of slices of the proxy geometry, without having to increase the size of the volume stored in memory.

2.2. Ray Casting Approach

Ray casting is a well-known volume rendering algorithm designed by Kajiya and Herzen [Kajiya 84] back in the 1980s. The basic idea is to trace rays from the camera into the volume, computing the volume rendering integral along the rays (see Figure 3).

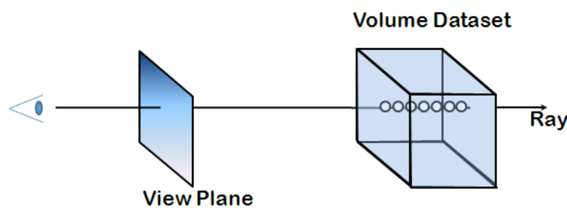


Figure 3: Ray casting scheme.

For each pixel in the image, typically a single ray is cast into the volume. The volume data is sampled at discrete positions along the ray. The contribution of each sample is accumulated to obtain the final color and opacity of the pixel.

This algorithm fits very well the GPU architecture and capabilities [Kruger 03, Scharsach 05]. In GPU ray casting, the volume data is uploaded to the GPU memory as a 3D texture. A fragment shader program is used to implement the ray casting algorithm, working on the fragments generated by rendering a polygon covering the screen space occupied by the volume bounding box. For each fragment of the polygon a ray is cast. Due to the independence between the per ray operations, and to the parallel architecture of the GPU, this operation can be done in parallel. The samples along the ray are taken using the hardware trilinear interpolation, and composed to compute the final pixel color.

2.3. Texture Based vs Ray Casting Approaches

Compared with the ray casting technique, the texture based approach has several drawbacks [Hadwiger 06, Scharsach 05]. First, it performs the evaluation of the volume rendering integral for fragments that do not contribute to the final image [Stegmaier 05, Kruger 03] (e.g. occluded fragments). This characteristic significantly increases the amount of texture fetch operations, numerical operations, i.e. lighting calculations, and per pixel blending operations that are executed. Due to the inflexible nature of this algorithm, advanced acceleration techniques which could correct this situation are hardly implementable.

Ray casting, on the other hand, is a much more flexible algorithm which allows for the integration of acceleration techniques that can solve the problem of unnecessary per fragment calculations [Kruger 03, Stegmaier 05]. The early ray termination mechanism which truncates the ray when the upcoming samples do not influence the final result, and the empty space skipping technique [Kruger 03] which skips volume regions that are considered empty are examples of these techniques.

Another disadvantage of the texture based approach compared to the ray casting technique is when a perspective view is applied: using a texture based approach and a perspective view, the sampling distances vary from ray to ray, introducing incoherences in the final image. In contrast, ray casting maintains a constant sampling distance, therefore avoiding visual artifacts [LM99].

For these reasons, the texture based techniques can be considered secondary for the implementation of a volume rendering framework, as they fail to take full advantage of the hardware capacities.

3. RAY CASTING IN GPU

The basic idea of GPU ray casting is to implement the ray casting algorithm in a fragment shader program. Multipass ray casting is an approach used to implement a GPU ray caster. It was described in [Kruger 03], by Krüger et al., and consists of the following main steps:

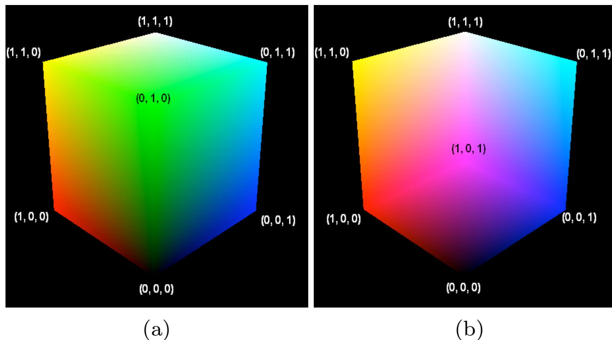
- a **first rendering pass**, processed in the GPU, in which the exit point for each ray is calculated
- a **second pass**, in which the entry point for each ray is obtained
- **main passes 3 to N** consist of sampling the volume dataset along the ray and combine the samples to determine the pixel color. In each pass, M steps along the ray are performed, and then an intermediate pass is executed
- **intermediate passes 3 to N,** where the stop criterion is tested and the ray is terminated in case it left the volume dataset boundaries or if

the opacity accumulated for the current pixel has reached a given threshold

This multipass approach was first designed to overcome hardware limitations, since early GPUs did not provide loops functionality and conditional branches were hard to implement [Hadwiger 06]. Therefore, the traversal of a ray was initiated and driven by a CPU-based program. Currently, loops and conditional branches are available in the instruction set of the GPU programming languages, which permits the simplification of the algorithm to two passes:

- in a **first pass**, the exit points for each ray are calculated
- in the **second pass**, the entry points of each ray are calculated, and using a loop instruction the rays traversal is performed, combining the samples collected, until the stop criterion is reached

The strategy to calculate the entry and exit points of the rays is to store the volumetric dataset in a 3D texture and define a bounding box for this dataset. This bounding box is a cube where the color channel encodes the 3D texture coordinates of the volumetric dataset boundaries (which range from 0 to 1). Rendering the front faces of the cube yields an image with the entry position of the rays in the bounding box, encoded in color (see Figure 4a). Drawing the back faces of the bounding box results in an image encoding the exit position of the rays (see Figure 4b).



(a) (b)

Figure 4: Rendered front (a) and back (b) faces of the bounding box.

Subtracting the two images shown in Figure 4, yields the lines going from the start to the exit ray positions. These lines are used to calculate the ray directions, necessary for the ray traversal step.

In the first pass of the algorithm, the back faces of the bounding box are rendered using the OpenGL fixed functionality. The resulting image is stored in a 2D texture (i.e. the exit texture). In the second pass, with the ray casting shaders enabled, the front faces of the bounding box are rendered. This will cause the GPU to receive the information of the entry points of the rays, in the form of a color for each pixel. The GPU combines this information with the exit points previously stored in a 2D texture and performs the ray set up and ray traversal, evaluating the color for each pixel. Following is a detailed description of the implementation.

3.1. Multipass Ray Casting Implementation

Algorithm 1 shows the core of the CPU part of the multipass ray casting algorithm.

Algorithm 1 Multipass ray casting algorithm in the CPU.

For each frame:

1. Render the back face of the bounding box to a 2D texture using the OpenGL fixed functionality
2. Enable the Ray Casting Shaders
3. Render the front face of the bounding box and perform the Ray Casting using the Ray Casting Shaders
4. Disable the Ray Casting Shaders
5. Enable the OpenGL fixed functionality

It consists of a loop where, for each frame, the back face of the volume bounding box is rendered to a 2D texture using the OpenGL fixed functionality. Then, with the ray casting fragment shader enabled (further described in Algorithms 2 and 3), the front face of the bounding box is rendered. This way, an interpolated color for each pixel becomes available in the fragment shader, corresponding to the texture coordinates of the ray starting point. The shaders enter in action to calculate the color value of each pixel, sampling the volumetric dataset. Finally, the OpenGL fixed functionality is re-enabled, and the display loop can be repeated.

To perform the second pass of the algorithm, a fragment shader program was designed to drive the per pixel operations. The first task of the fragment shader is the ray set up. It consists in finding, for each pixel, the entry and exit points of the ray in the volume bounding box. This information, combined with the step size, allows to compute the ray direction, the ray length and the step vector needed for the ray traversal. The next step is to traverse the ray according to a given step size. The hardware built-in trilinear interpolation is used to obtain the value of each data sample from the 3D texture which stores the volume dataset. At last, the color for each pixel is calculated by compositing the samples obtained.

The structure of the fragment shader is divided in two parts. The first one, called the ray set up, is described in Algorithm 2.

For each pixel, the exit and entry points of the ray in the bounding box are fetched (instructions 1 and 2 respectively). In step 3, the line from the entry point

Algorithm 2 Ray Set Up in the fragment shader program.

```
Ray Set Up:

1. Get the ray exit position from the exit texture
     exitRayPosition = getValue(current pixel position,
                                exit texture);

2. Get the ray starting position from the color of the current pixel
     startRayPosition = currentPixelColor;

3. Compute the maximum ray length, which can be used to terminate the ray
     rayLine = exitRayPosition - startRayPosition;
     maxRayLength = length(rayLine);

4. Compute the step vector
     normalizedRay = normalize(rayLine);
     stepVector = normalizedRay * stepSize;
```

to the exit point is calculated (rayLine) and used to compute the maximum ray length (maxRayLength). The maximum ray length is used later in the ray traversal loop (described in Algorithm 3) to test whether to terminate the loop or not. Finally, in step 4, a step vector is calculated (stepVector). The step vector is used to increment the sampling position during the ray traversal. Its length is used to accumulate the total length traversed so far.

The second part of the fragment shader program, called ray traversal, is the one that actually "shoots" the ray, i.e. collects the samples and composites them into the final pixel color (see Algorithm 3).

It is assumed that the same amount of light reaches every point inside the volume. The first step (1) is to initialize the variables where the color and alpha values will be accumulated (accumulatedColor and accumulatedAlpha respectively). In step 2, the ray traversal loop starts. The loop starts by getting a volume sample using the 3D hardware built-in interpolation (step 2.1). In step 2.2, the value is classified using the transfer functions, yielding a color and an alpha values (colorSample and alphaSample). Notice that, to compute the volume rendering integral, the alpha value is multiplied by the step size. That is because the opacity value to be summed to the accumulated alpha depends on the sampling distance (given by stepSize). The volume rendering integral is updated in 2.3: the alpha value of the current sample is summed to the accumulated alpha value, and the sampled color is incorporated in the accumulated color value. The sampling position and the traversed ray length are refreshed in steps 2.4 and 2.5 respectively. This loop will be repeated till the ray exceeds the previously evaluated maximum ray length, or till the maximum opacity value is reached (see the loop condition in step 2). The second loop condition (accumulatedAlpha < 1.0) implements the early ray termination mechanism, a feature of the ray casting algorithm which consists on truncating the light rays as soon as the volume elements further away along the ray are occluded. Finally, after the loop termination, the accumulated color and opacity are displayed (step 3).

Algorithm 3 Ray Traversal in the fragment shader program.

```
Ray Traversal:

1. Initialize accumulation variables
   accumulatedColor = (0.0, 0.0, 0.0);
   accumulatedAlpha = 0.0;

2. Ray traversal loop
   while(currentRayLength < maxRayLength &&
         accumulatedAlpha < 1.0)

      2.1. Get a volume sample
           sample = getSampleValue(volume texture,
                                   current ray position);

      2.2. Get the optical properties for the sample
           colorSample = getColorValue(color transfer function,
                                       sample);
           alphaSample = getAlphaValue(opacity transfer function,
                                       sample) * stepSize;

      2.3. Update the Volume Rendering Integral
           accumulatedColor += (1.0 - accumulatedAlpha) *
                               (colorSample * alphaSample);
           accumulatedAlpha += alphaSample;

      2.4. Compute the next sample position
           currentRayPosition += stepVector;

      2.5. Compute the ray length traversed
           currentRayLength += stepSize;

3. Attribute the accumulated color and opacity to the pixel
   pixelColor = accumulatedColor;
   pixelAlpha = accumulatedAlpha;
```

3.2. Jittering

The use of a large step size which allows for a faster rendering time, can cause aliasing in the final image, which results in visible artifacts named wood-grain effects. Stochastic jittering is a technique used to hide wood-grain effects by introducing a variation in the starting position of the rays, along the viewing direction [Hadwiger 06]. This causes the aliasing to be substituted by noise.

The variation introduced causes the samples along the ray to be offset by a random number ranging from 0 to the step size value. The samples along a ray have the same offset, while different rays are likely to have assigned a different jitter value. Consequently, the coherence between pixels which causes wood-grain artifacts is suppressed by noise.

The implementation of the jittered multipass ray casting does not differ much from the multipass ray casting algorithm described in the previous section. The CPU part of the algorithm, differs in that a 2D texture with size 32×32 is created, containing a random number at each position. This texture is uploaded to

the graphics card memory, during the application set up phase, and is later used by the fragment shader as a source of random numbers to perturb the starting positions of the rays.

In the fragment shader program, only the ray set up stage differs from the implementation described in section 3.1. Here, a variation ranging from 0 to the current step size value is introduced in the ray starting position. This value is calculated based on the 2D texture holding the random numbers, and added to the ray starting position. The ray traversal stage remains unchanged. Algorithm 4 shows the ray set up stage for the jittered multipass ray casting.

Algorithm 4 Ray Set Up for the jittered multipass ray casting.

```
Ray Set Up:

1. Get the ray exit position from the exit texture
     exitRayPosition = getValue(current pixel position,
                                exit texture);

2. Get the ray starting position from the cube color already interpolated
     startRayPosition = currentPixelColor;

3. Compute the ray line
     rayLine = exitRayPosition - startRayPosition;

4. Compute the step vector
     normalizedRay = normalize(RayLine);
     stepVector = normalizedRay * stepSize;

5. Introduce an offset in the ray starting position along the ray direction
     offset = getRandomNumber(jitterTex,
                              exitFragPosition * textureSize));
     startRayPosition += offset * stepSize * normalizedRay;

6. Compute the maximum ray length, which can be used to terminate the ray
     maxRayLength = length(rayLine) - (offset * stepSize);
```

The sequence of steps 1 to 4 yields the normalized ray direction, the step vector and its length, necessary to perform the ray traversal. In step 5, an offset based on a random number extracted from the jitter texture is calculated and added to the ray starting position. The ray set up stage ends with the computation of the new ray length (step 6), and the ray traversal stage is ready to be executed.

3.3. Empirical Visualization Methods

The ray casting technique can be used for alternative visualization techniques which might be useful to understand the information contained in the 3D dataset, rather than to evaluate the volume rendering integral presented in Algorithm 3. Examples of these alternative techniques are the X-Ray and the Maximum Intensity Projection (MIP) compositing methods, often applied in medical imaging applications [Preim 07, Hadwiger 06]. These two techniques compute the final image in the following way:

- **X-Ray.** The scalar values of the data samples along each ray are summed up, resulting in a final image close to an X-Ray image.
- **Maximum Intensity Projection.** For each pixel, only the sample with the highest value along the ray is taken into account for the pixel color.

In Algorithm 5 the ray traversal scheme used to implement the X-Ray compositing method is presented. It is assumed that the ray set-up phase was already executed, yielding all the information necessary to perform the ray traversal. The ray set-up phase can be either jittered (Algorithm 4) or non-jittered (Algorithm 2). Algorithm 5 consists on summing up the scalar values for each sample along the ray. The opacity is directly attributed (in 2.3) from the scalar value acquired from the dataset in 2.1. The color is also directly attributed from the scalar value, yielding a gray scale where the most opaque values are colored in white (2.2). Lighting information is not considered, leading to a final image which is as it would be composed of X-Ray (see Figures 7a and 7b).

Algorithm 5 Ray Traversal for X-Ray compositing.

```
Ray Traversal:

1. Initialize accumulation variables for color and opacity at zero

2. Ray traversal loop
   while(currentRayLength < maxRayLegth &&
         accumulatedAlpha < 1.0)

      2.1. Get a volume sample
           sample = getSampleValue(volume texture,
                                   current ray position);

      2.2. Update the accumulated color
           accumulatedColor += sample;

      2.3. Update the accumulated opacity
           accumulatedAlpha += sample * stepSize;

      2.4. Compute the next sample position

      2.5. Compute the ray length traversed

3. Attribute the accumulated color and opacity to the pixel
```

Maximum Intensity Projection is a popular compositing mode that searches for the highest sample value along a ray. The main idea is to traverse the ray, and attribute the value of the highest sample found, in gray scale, to the pixel color. MIP is mostly used to display bone structures and contrast enhanced vascular structures (vessels), where the measured intensity is significantly higher than the regular tissue value [Preim 07, Hadwiger 06]. Algorithm 6 presents the ray traversal for the MIP compositing method.

The traversal loop is executed until the ray is completely traversed (step 2.). For each loop, the volume

Algorithm 6 Ray Traversal for Maximum Intensity Projection.

Ray Traversal:

1. Initialize the variable holding the highest sample value
   maxSample= 0.0;

2. Ray traversal loop
   while(currentRayLength < maxRayLegth)

   2.1. Get a volume sample

   2.2. Store the current sample value if it has the highest value so far
      if (sample > maxSample)
         maxSample = volumeDataSample;

   2.3. Compute the next sample position

   2.4. Compute the ray length traversed

3. Attribute the color in gray scale, and set the opacity to 1 (maximum)
   pixelColor = (maxSample, maxSample, maxSample);
   pixelAlpha = 1.0;

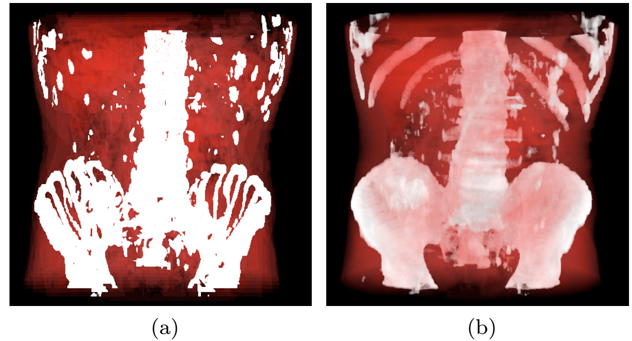
is sampled at the current position (2.1). If the sample value is higher than the maximum sample value taken so far, the variable maxSample is refreshed with the current sample value (2.2). The sample position is incremented to the next position on the ray, and the total ray length traversed so far is computed (2.3 and 2.4). At last, after the loop termination, the accumulated color and opacity are attributed to the pixel, in step 3. (see Figures 7c and 7d).

4. RESULTS AND DISCUSSION

A dataset with a dimension of $512 \times 512 \times 246$ (64.487.424 voxels) has been rendered to a 1000×1000 viewport in a machine equipped with an *ATI Radeon HD 3450 GPU* and *OpenGL 2.1*, using three different opacity transfer functions. Each of the transfer functions, when applied to a given dataset sample, yields a different opacity level (high, medium and low). The shaders were implemented using the *GLSL* language.

Figure 5 shows the variation of the quality of the final image according to the step size. It can be observed that the quality of the final image is dependent on the step size. Using a big step size results in an undersampled, strongly aliased image, with the wood-grain effect referred in section 3.2. As the step size is decreased, and consequently, the number of samples per ray increases, the rendered image demonstrates a higher quality. Figure 5b shows this behavior. However, as it can be seen in Table 1 high quality may cause loss of interactivity.

Table 1 shows the average framerate achieved, with 4 different step sizes, for each of the three volume opacities tested, when a complete rotation is applied to the



(a) (b)

Figure 5: Images obtained for different step size values. In (a) a step size of 0.050, resulting in a maximum of 34 samples per ray was used. Image (b) was rendered with a step size of 0.005, yielding a maximum of 346 samples per ray.

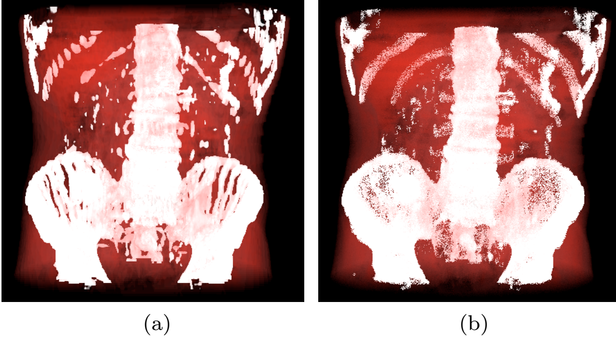
	Step Size			
	0.125	0.050	0.025	0.005
low op.	43	24	13	4
medium op.	43	25	14	4
high op.	45	28	16	5

Table 1: Rendering speed in frames per second achieved for the three volumes rendered, and different step sizes.

dataset. The results show that for small step sizes, the rendering speed decreases dramatically. This can be seen by comparing the 25 fps framerate for the medium opacity volume with a step size of 0.050, with the 4 fps yielded for a step size of 0.005. As expected, the results also demonstrate that the number of frames per second (fps) increases with more opaque volume datasets, due to the early ray termination mechanism implemented.

The jittering technique was implemented as a possible solution for achieving higher quality results without decreasing the step size, by introducing a random variation in the sampling positions. Figure 6 shows the result of rendering the dataset with medium opacity and a step size of 0.025, with multipass ray casting and jittered multipass ray casting.

The image resulting from rendering with a stochastic jitter (Figure 6b) does not contain the regular patterns (wood-grain effects) which can be observed in the original image (Figure 6a). The patterns are substituted by noise caused by the random variation introduced in the ray starting positions. For a human being, the noise is easier to tolerate than the regular patterns present in the non jittered image. As a conclusion, the result obtained with the jittered multipass ray casting is visually more acceptable.



(a) (b)

Figure 6: Comparison of the images obtained when rendering the volume dataset, with medium opacity, with multipass ray casting (a) and with jittered multipass ray casting (b). A step size of 0.025 was used.

Due to the increase of the per fragment operations, rendering a volume with jittered multipass ray casting is slower than using the multipass ray casting technique for the same step size. In Table 2 a comparison of the average framerate for the medium opacity volume, obtained when applying a complete rotation to the dataset, can be seen. Different step sizes were used, and combined with jittered and non jittered ray set up.

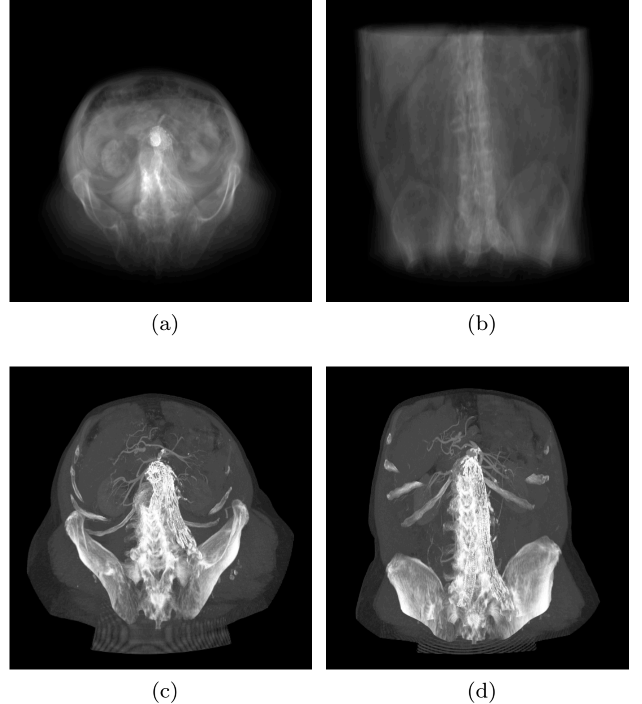
	Step Size			
	0.125	0.050	0.025	0.005
jittered	23	16	11	4
non jittered	43	25	14	4

Table 2: Comparison of the fps obtained when rendering the medium opacity volume with jittered multipass ray casting (jittered) and with multipass ray casting (non jittered), for different step sizes.

The results in Table 2 show that the jittered version is consistently slower than the non jittered version. But the decrease of the number of fps for the jittered version is not too high regarding the corresponding improvement in the subjective image quality. For example, for the images shown in Figure 6 (rendered with a step size of 0.025), the difference between the two versions is of 3 fps. As the step size decreases, the ray set up overhead introduced in the jittered multipass ray casting becomes less relevant for the rendering time and the results obtained with both techniques converge (see for example the results for a step size of 0.005, in Table 2).

Figure 7 shows the results for rendering the dataset with X-Ray and MIP techniques.

Figures 7a and 7b were rendered using the X-Ray technique. The weighted sum of the samples along the ray results in a final image similar to an X-Ray image. In the Figures 7c and 7d, rendered using the MIP compositing technique, only the highest sampled value along each ray contributes to the final image. This emphasizes the bone and vascular structures present in the dataset, but does not provide the viewer with depth information.



(a) (b)

(c) (d)

Figure 7: Images (a) and (b) show the dataset rendered with the X-Ray compositing technique. In (c) and (d), the dataset was rendered with the MIP compositing method.

In Table 3, the average framerate obtained while rendering a complete rotation of the dataset, using the X-Ray and MIP compositing methods, is shown.

	Step Size			
	0.125	0.050	0.025	0.005
X-Ray	28	19	16	7
MIP	29	21	18	7

Table 3: Comparison of the time per frame obtained while rendering the volume dataset with the X-Ray and MIP compositing methods, with jittered ray set up, and different step sizes.

The results show that the performance of these techniques is in line with the results achieved previously (e.g. see Table 2, jittered), with an increasing rendering time as the step size decreases. The simplicity of the operations during the ray traversal, cause the time to render an image with the X-Ray and MIP compositing methods to be less than the one needed to evaluate of the volume rendering integral (e.g. Table 2, jittered).

5. CONCLUSIONS

In this article, a set of shaders used to perform volume ray casting on the GPU was presented. Comparisons were performed among different compositing techniques, and the effect of different step sizes along each ray was also evaluated. Larger step sizes result in faster rendering, but aliasing becomes apparent in the resulting images. Jittering of the origin of the rays was introduced to minimize this problem, using this stochastic process to trade noise for aliasing. Noise is more easily tolerated by the Human Visual System than the visible artifacts caused by the aliasing, which allows for the utilization of larger step sizes achieving the same subjective image quality.

This project was motivated by the creation of a Human Atlas visualization tool, based on volume rendering techniques, which could allow a real time interaction. The large rendering times obtained by using traditional CPU ray casting, prohibitive for a real time visualization, and the availability of extremely efficient and highly programmable GPUs, drove the project to the field of GPU ray casting. The results achieved so far are satisfactory regarding both image quality and rendering time. The parallel nature of the ray casting algorithm, where each ray is processed independently of the other rays, sugests that the shaders implemented should be well scalable.

References

[Bruckner 08] Stefan Bruckner. *Efficient Volume Visualization of Large Medical Datasets - Concepts and Algorithms.* VDM Verlag, Saarbrücken, Germany, 2008.

[Cabral 94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *Proc. of the 1994 symp. on volume visualization*, pages 91–98, 1994.

[Cullip 94] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3d texture hardware. *Technical Report: Univ. of North Carolina at Chapel Hill*, 1994.

[Hadwiger 06] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics.* A. K. Peters, Ltd., USA, 2006.

[Johnson 04] Christopher Johnson and Charles Hansen. *Visualization Handbook.* Academic Press, Inc., USA, 2004.

[Kajiya 84] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, 1984.

[Kruger 03] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. *Proc. of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, 2003.

[Levoy 88] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[LM99] Eric C. La Mar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. *Proc. of the 10th IEEE Visualization 1999 Conf. (VIS '99)*, 1999.

[Preim 07] B. Preim and D. Bartz. *Visualization in Medicine: Theory, Algorithms, and Applications.* Elsevier, 2007.

[RS00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. *Proc. of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118, 2000.

[Scharsach 05] H. Scharsach. Advanced gpu raycasting. *Proc. of CESCG*, pages 69–76, 2005.

[Stegmaier 05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. *Volume Graphics, 2005. Fourth Int. Workshop*, pages 187–241, June 2005.

[Weiskopf 07] Daniel Weiskopf. *GPU-Based Interactive Visualization Techniques (Mathematics + Visualization).* Springer-Verlag Berlin Heidelberg, 2007.

[Westermann 98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. *Proc. of the 25th Annual conf. on Computer Graphics and Interactive Techniques*, pages 169–177, 1998.