ISSN 1213-6972 Journal of WSCG
http://www.wscg.eu Vol.27, No.1, 2019

# Fast Adaptive Undersampling for Volume Rendering

Sergey Belyaev
Peter the Great St. Petersburg Polytechnic University
Polytechnicheskaya 29
195251 St. Petersburg, Russia
bel_s_u@mail.ru

Pavel Smirnov
Peter the Great St. Petersburg Polytechnic University
Polytechnicheskaya 29
195251 St. Petersburg, Russia
psmirnov@spbstu.ru

Natalia Smirnova
EPAM Systems, Inc.
41 University Drive, Suite 202
Newton, PA 18940, USA
Natalia_Smirnova@epam.com

Vladislav Shubnikov
Peter the Great St. Petersburg Polytechnic University
Polytechnicheskaya 29
195251 St. Petersburg, Russia
vlad.shubnikov@gmail.com

## ABSTRACT

Adaptive undersampling is a method for accelerating the rendering process by replacing the calculation of a volume integral with an interpolation procedure for a number of pixels. In this paper, we propose a method for accelerating the volume integral calculation for the rest of the pixels, i.e. those pixels for which interpolation cannot be done with sufficient accuracy. This method requires two passes through the input data. On the first pass, rendering is done into a low-resolution texture. At this stage, the values of the volume integral on a set of intervals of a given length are calculated and saved into a special G-buffer alone with the pixel's color. On the second pass, these values are used to determine colors of the pixels. For those pixels whose result is not precise enough, the volume integral is calculated on one or several intervals, rather than the whole ray. The proposed method allows one to accelerate adaptive undersampling by a factor of 1.5 on average, depending on the input data.

## Keywords

Volume rendering, Ray casting, Adaptive sampling.

## 1. INTRODUCTION

The main visualization method for volumetric scientific data (e.g. medical data) is direct volume rendering, which calculates the value of the volume integral for each screen pixel. This approach uses scanning of the large volumes of data efficiently using various transfer functions, but this process is computationally expensive. Its running time is proportional to the number of pixels in the visualization window, so its optimization for high-resolution screens and devices with low computing power is a relevant problem. Examples of such devices include mobile phones, laptops and PCs with slow video cards, as well as VR devices, which require a minimum of 60 FPS while rendering into two cameras at the same time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

The method is usually implemented on GPUs in conjunction with various optimization techniques—discarding regions on which the transfer function is zero [LCDP12], varying the integration step [CCF15], pre-integrated volume rendering [KE04], and adaptive undersampling (or screen undersampling) [KRHH11]. Adaptive undersampling makes use of the coherency of the scene in order to minimize the number of volume integrals to be calculated to determine the color of pixels in the image. This is achieved by an iterative procedure. On the first iteration, only part of the pixels is sampled (one for each $n \times n$ block), and then an attempt is made to recover the colors of the rest of the pixels with the information thus obtained (for example, by interpolating bilinearly between the colors of adjacent pixels). If this does not produce the required image quality, then the set of pixels being sampled is expanded. In practice, most input data sets (including medical data) have high levels of spatial coherence, which means that after the first iteration, only around

10% to 20% of the total number of pixels need to be sampled additionally. Unfortunately, when the algorithm is implemented on a GPU, additional calculations for some pixels lead to increase of the processing time for all pixels of the image, as if the optimization is completely absent. This is because the pixels are processed concurrently in groups, and the time it takes to process a group is equal to the maximum of the times to process each pixel. Thus, if at least one pixel in the group calls for the calculation of a volume integral, then the whole group will take exactly as much time to process as if every pixel's volume integral had to be calculated.

In this paper, we propose a two-pass algorithm which solves the problem by making the calculation of the volume integrals on the second pass much faster for pixels whose colors cannot be interpolated. To do this, on the first pass, the domain of integration for the volume integral is broken up into $M$ pieces, and the values of the integral on each piece are saved into the G-buffer. On the second pass, the volume integral is calculated by summing its values on $M$ pieces. These values are determined either by bilinearly interpolating the corresponding G-buffer values, or, if that is not possible, by integration.

Below is an overview of related work (Section 2), followed by a discussion of what we consider to be our main contribution: a method for accelerating volume rendering by pre-computing the volume integrals on multiple intervals for part of the pixels (Section 3). We discuss the results in Section 4 and make conclusions in Section 5.

## 2. RELATED WORK

The most flexible and widespread method for direct volume rendering is raycasting. GPU-based raycasting was proposed in [KW03]. It uses cube proxy geometry (the bounding box of the dataset) to determine the starting and ending points of the way. However, the method is slow, as it requires the volume integral to be calculated for every pixel by going down the whole ray from start to end with some step. Adaptive sampling can be used for raycasting optimization. This allows to obtain the output image by calculating the volume integral for only part of the pixels. This was first proposed in [Lev90], in which the volume integral is calculated in the corners of equally sized blocks into which the image is partitioned. If the values in these corners do not differ significantly, then the colors of the interior pixels of the block are interpolated bilinearly. Otherwise, the block is partitioned into four parts, and the procedure is applied recursively to each part. Kratz et al. [KRHH11] present a variation of Levoy's approach for GPU-based rendering. They replaced the comparisons of the integrals at the blocks' corners with a more sophisticated technique based on finite element methods (FEM) to achieve explicit error control. In their implementation, raycasting was done on the GPU, while the hierarchical data structures of the blocks (quadtree) were stored on the CPU. [KSK*16] and [BSSS18] examine methods for excluding artifacts which arise due to the fact that volume integrals are not calculated for all pixels.

On a GPU, recursive division leads to multi-pass algorithms which turn out inefficient due to the architecture of a GPU. Thus, [L15] uses a two-pass algorithm, in which the colors of interior pixels are calculated either via bilinear interpolation or by calculating the volume integral. The two-pass algorithm is also used in [BFE16] in order to optimize raytracing on mobile devices.

In [BSM18] the second rendering pass is accelerated by saving (on the first pass) volume integral values in the ray intervals, where the transfer function value is not zero. Unfortunately, this algorithm is effective only when number of intervals is relatively small.

## 3. ALGORITHM

### 3.1. Overview

The volume integral for each pixel gives the fraction of light passing through the volume along the pixel's view ray. The discrete form of this integral can be efficiently computed via compositing, which replaces a Riemann sum with a recurrence relation:

$$C_{i+1} = C_i + (1 - A_i) \cdot a_i \cdot c_i. \quad (1)$$
$$A_{i+1} = A_i + (1 - A_i) \cdot a_i.$$

In the above equations, $C_i$ is the composited color on the $i$'th step along the ray, $A_i$ is the composited transparency, and $a_i$ and $c_i$ are, respectively, the transparency and color in the given sample.

The proposed algorithm is based on two-pass adaptive undersampling. The set of pixels is partitioned into $n \times n$ blocks, and on the first pass one pixel from each block is processed. However, unlike the method above, our algorithm divides the interval of integration into $M$ equal pieces, and the values of the integral over these pieces are saved into the G-buffer along with the color of the pixel.

More details concerning $M$ value will be explained in section 4. In Figure 1, which depicts the

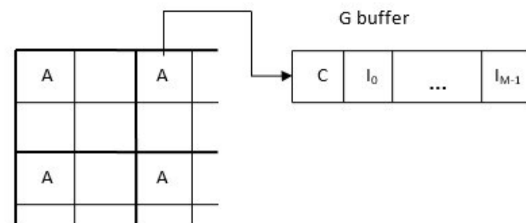



Figure 1: The G-buffer contains the color ($C$) of the pixel and the values ($I_i$) of the volume integral on a set of equal length intervals.

case $n = 2$, the pixels being processed are marked with an A. The volume integral over the $i$'th interval is denoted with $I_i$ ($i < M$). $C$ and $I_i$ are 4-component vectors, containing the colors $C_i$ and transparencies $A_i$. Colors $C_i$ contain three components: red, green, blue.

The rest of the pixels are processed on the second pass. If the colors of their adjacent pixels are sufficiently close, then the color $C^P$ of the current pixel is interpolated bilinearly. Otherwise, it is calculated with the following recurrence relation, according to [HLSR09]:

$$C^P{}_{i+1} = C^P{}_i + (1 - A^P{}_i) \cdot A_i^* \cdot C_i^*, \qquad (2)$$
$$A^P{}_{i+1} = A^P{}_i + (1 - A^P{}_i) \cdot A_i^*, \quad 0 \le i \le M - 1.$$

In the above equations, $A_i^*$ and $C_i^*$ are interpolated bilinearly from the values of $A_i$ and $C_i$ in the adjacent pixels if those values are close enough and are calculated from the volume integral otherwise. Thus, on the second pass the volume integral is only calculated over the part of the ray in the worst case, which significantly accelerates the generation of the whole image.

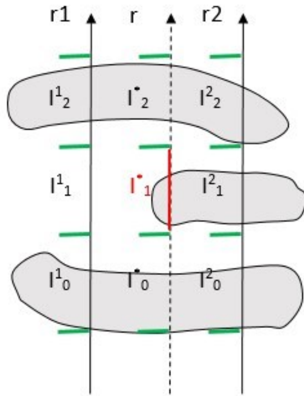



Figure 2: The volume integral only needs to be calculated on the red interval.

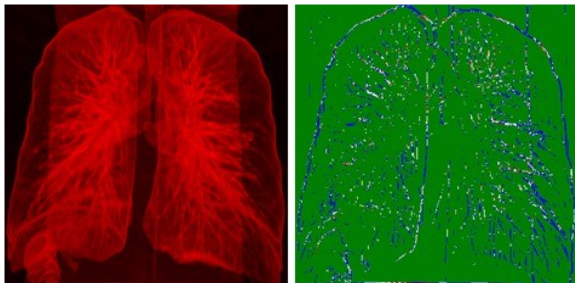


Figure 3: Left: the resulting image. Right: the pixels whose volume integrals were interpolated bilinearly are shown in green (91.7%); those for which the volume integral had to be calculated over one interval are shown in blue (5.7%); over two intervals, in white (2.1%); over three intervals, in yellow (0.1%); and more than three intervals in red (0.4%).

The algorithm accelerates rendering by reducing the length of the interval of integration. In Figure 2 (drawn in two dimensions for simplicity), an example is shown for the second pass of the algorithm for the case $M = 3$, where $r_1$ and $r_2$ denote the rays passing through pixels processed on the first pass. The integrals $I_i^1$ and $I_i^2$ have been calculated and are stored in the G-buffer. The current pixel being processed is on the ray $r$. The values of $A_i^*$ and $C_i^*$ for $i = 0, 2$ are interpolated from $I_i^1$ and $I_i^2$, while $A_1^*$ and $C_1^*$ are calculated via the volume integral $I_1^*$ on the given interval. Figure 3 shows the number of pixels in a real dataset for which the volume integral needs to be calculated on the second pass, and the number of intervals on which it must be calculated. The pixels whose volume integrals were interpolated bilinearly are shown in green. Those for which the volume integral had to be calculated are colored based on how many intervals it had to be calculated on: blue for 1, white for 2, yellow for 3 and red for more than 3. As can be seen from the figure 3, in most cases the integral only needed to be calculated over one interval, which is what makes the algorithm so efficient. The following is a detailed description of the algorithm.

## 3.2. Algorithm details

In the first pass, the algorithm fills the $M$ parallel textures (in the G-buffer) which have a resolution $n$ times less than the viewport (along each side). Algorithm 1 shows a pseudocode for each ray calculation. The function *GetDistanceForStart*($s$, $f$) called in line 1 finds the distance from the starting point $s$ of the ray to the point $v$ where it first meets the domain where the transfer function is not zero (Figure 4). Here, $f$ is a final point on ray, both $s$, $f$ are 3D vectors, step initialization is explained below.

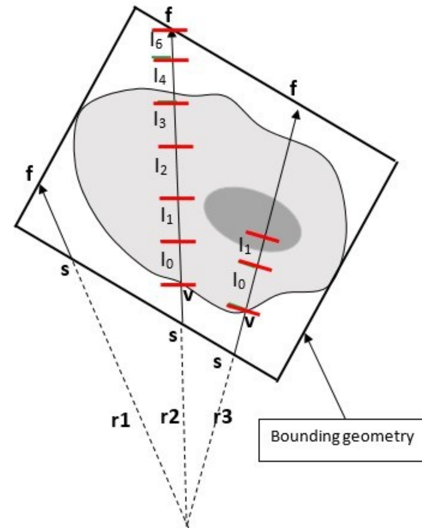



Figure 4: No integration is done over $r_1$. Over $r_2$ and $r_3$ integration begins at the points marked with $v$.

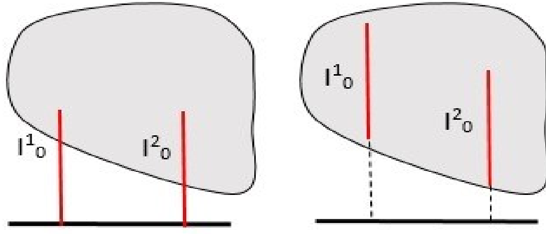



Figure 5: On the left, $I^1_0 \neq I^2_0$; on the right, $I^1_0 = I^2_0$.

Condition ($A < 1$) in line 18 means that total opacity has not reached 1, i.e. integration process along ray has not stopped at this point. Important detail: last integration calculation has another "*lastStep*", not equal to "*step*" and will be explained in detail below. *VolumeIntegral*($v$, *step*) is a function, calculating the partial sum along a ray, starting from point $v$ with "*step*" length. We find the point $v$ for two reasons. First, the probability that the first integrals $I_0$ coincide on adjacent rays increases (see figure 5). Second, this helps remove "woodgrain" artifacts, especially in cases where the derivative of the opacity function is high in a neighborhood of $v$. This is explained in more detail in [LJKY13]. If $v$ is not found, the function returns −1 and the algorithm halts (as in the case of the ray $r_1$ in Figure 4). Otherwise, the algorithm calculates the color of the pixel and the volume integral over intervals of equal length ($r_2$).

All calculations are done in the texture space of the 3D texture which stores the data to be visualized. All samples are contained in a cube with sides equal to 1, so the longest ray in the texture space has length $\sqrt{3}$ (the diagonal of the cube). This value is used to calculate the interval length in line 6, where $M$ is the user-selected maximum number of intervals. The integrals are calculated in line 12 and are stored in the G-buffer; the integration itself can be done using any known method. The color of the pixel is stored in $G$[0].rgb in line 13. As can be seen from Figure 4, the last interval of integration can be shorter than the rest; this interval is processed in lines 18–21.

On the second pass, the colors are calculated for those pixels which were not processed on the first pass. Shown below is the Algorithm 2 that does this. It uses data from the G-buffer which was created on the first pass for the four neighboring pixels. In line 2, the current pixel's color is interpolated bilinearly if the neighbors' colors are sufficiently close.

Index $i$ in $G_i$ means neighborhood texel, calculated on the first pass. Index $i$ can be in range [0…3], due to four neighborhood texel for current ray, calculated during second pass. Condition for simple bilinear interpolation is based on comparison maximum color difference for neighborhood pixels with some parameter *delta*.

**Algorithm 1** The first pass algorithm

```
1:  G[0].a = GetDistanceForStart(s, f);
2:  if G[0].a ≤ 0 then
3:    G[0].rgb = BackgroundColor;
4:  else
5:    G = 0, i = 0, C = 0, A = 0;
6:    step = √3 / M;
7:    imax = floor(length(f − s) / step);
8:    lastStep = length(f − s) − step ∗ imax;
9:    r = normalize(f − s);
10:   v = s + G[0].a · r;
11:   while A < 1 and i ≤ imax do
12:     G[i + 1] = VolumeIntegral(v, step);
13:     C = C + (1 − A) · G[i + 1].rgb ×
            × G[i + 1].a;
14:     A = A + (1 − A) · G[i + 1].a;
15:     v = v + step · r;
16:     i = i + 1;
17:   end while
18:   if A < 1 then
19:     G[imax + 1] = VolumeIntegral(v, lastStep);
20:     C = C + (1 − A) · G[imax + 1].rgb ×
            × G[imax + 1].a;
21:   end if
22:   G[0].rgb = C;
23: end if
```

**Algorithm 2** The second pass algorithm

```
1:  if for all i,
         maxj||Gj[0].rgb|−|Gi[0].rgb|| < delta then
2:    C = BilinearInterpolation(Gi.rgb);
3:  else
4:    A = 0, C = 0;
5:    t = r · min{Gi[0].a};
6:    v = s + r · GetDistanceForStart(s + t, f);
7:    for k = 1 … M do
8:      if for all i,
             maxj||Gj[k].rgb|−|Gi[k].rgb|| < delta/M then
9:        I = BilinearInterpolation(Gi);
10:     else
11:       if k = M then
12:         Length = step;
13:       else
14:         Length = lastStep;
15:       end if
16:       I = VolumeIntegral(v, Length);
17:     end if
18:     C = C + (1 − A) · I.rgb · I.a;
19:     A = A + (1 − A) · I.a;
20:     v = v + step · r;
21:   end for
22: end if
```

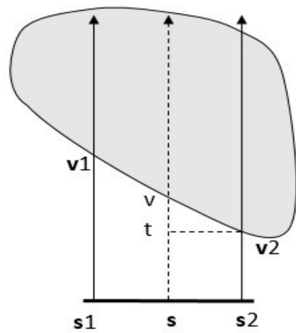


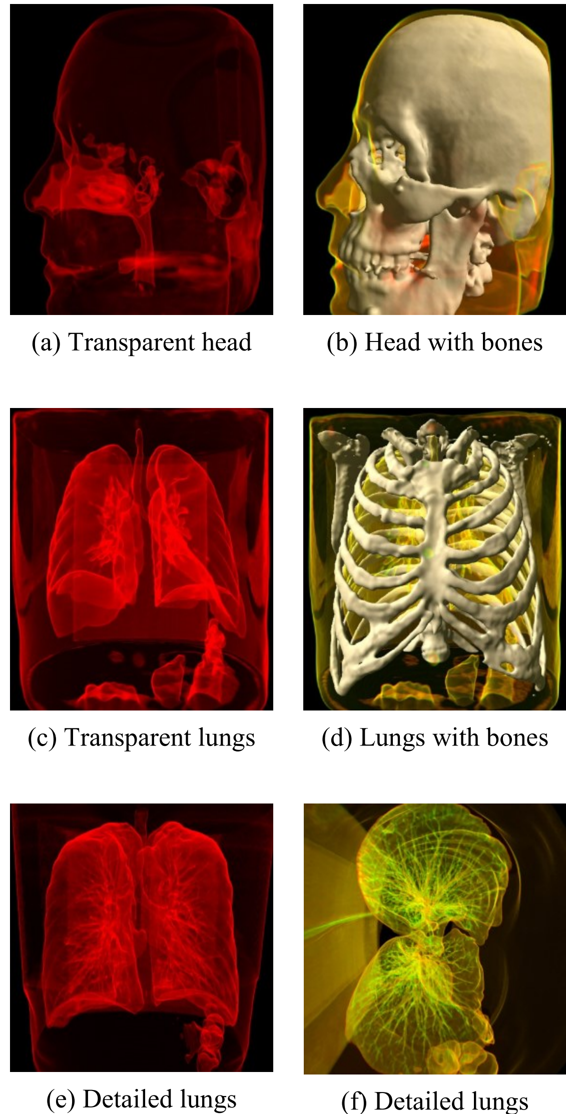

Figure 6: Calculating the starting point for integration.

In line 6, the algorithm finds the first point along the ray where the transfer function is not zero. This uses the same *GetDistanceForStart* function as in the first pass, but in order to accelerate its execution the ray is cast from $s + r \cdot \min\{G_i[0].a\}$, rather than from $s$ (see Figure 6). The loop (lines 7–21) implements the recurrence relations in formula 2. The integral calculation function in line 16 coincides with the function used in the first pass.

## 4. RESULTS AND DISCUSSION

All tests were performed on a 3.4GHz Intel Core i7 2600 PC with 4.0GB of main memory with NVidia GForce GTX 780 Ti graphics hardware with 3072MB of texture memory and implemented using Unity3D, using OpenGL ES 3.1. Three CT data sets were used as testing data; their characteristics and screenshots are given in Figure 7.

Table 1 contains the framerate achieved in visualizing the data sets. The volume integrals were calculated using the standard method [KW03] with ¼ of the voxel size as the step size. The bounding volume was chosen to be a box. The viewport was 1200×900 pixels. The value of $M$ was chosen as 8, which is the maximum possible size of the G-buffer on the video card used. The value of delta (see Algorithm 2) was chosen as 0.05.



(a) Transparent head (b) Head with bones

(c) Transparent lungs (d) Lungs with bones

(e) Detailed lungs (f) Detailed lungs

Figure 7: Data sets; resolution is 256×256×256 for (a)–(d) and 512×512×136 for (e)–(f).

| Data Set | OpenGL ES 3.1 FPS | | | B/A | C/A | C/B |
|---|---|---|---|---|---|---|
| | A | B | C | | | |
| (a) | 32 | 72 | 118 | 2.25 | 3.69 | 1.64 |
| (b) | 48 | 92 | 116 | 1.92 | 2.42 | 1.26 |
| (c) | 34 | 68 | 112 | 2.00 | 3.29 | 1.65 |
| (d) | 40 | 66 | 74 | 1.65 | 1.85 | 1.12 |
| (e) | 18 | 32 | 50 | 1.78 | 2.78 | 1.56 |
| (f) | 20 | 42 | 74 | 2.10 | 3.70 | 1.76 |

Table 1: The framerate achieved in visualizing the data sets.

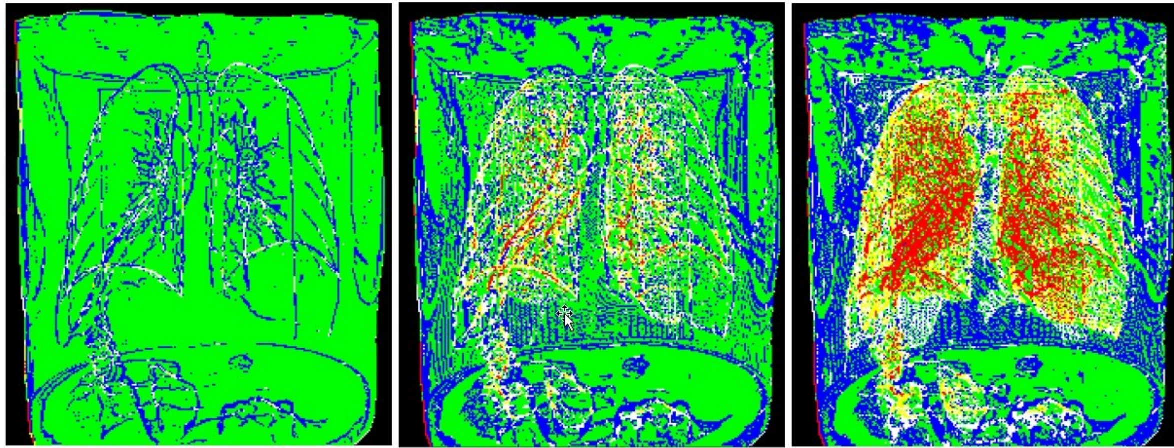


Figure 8: The number of intervals of integration per pixel for delta = 0.05, 0.01, 0.005 (left to right). Color key: green = 0 (bilinear interpolation), blue = 1, white = 2, yellow = 3, red = more than 3.

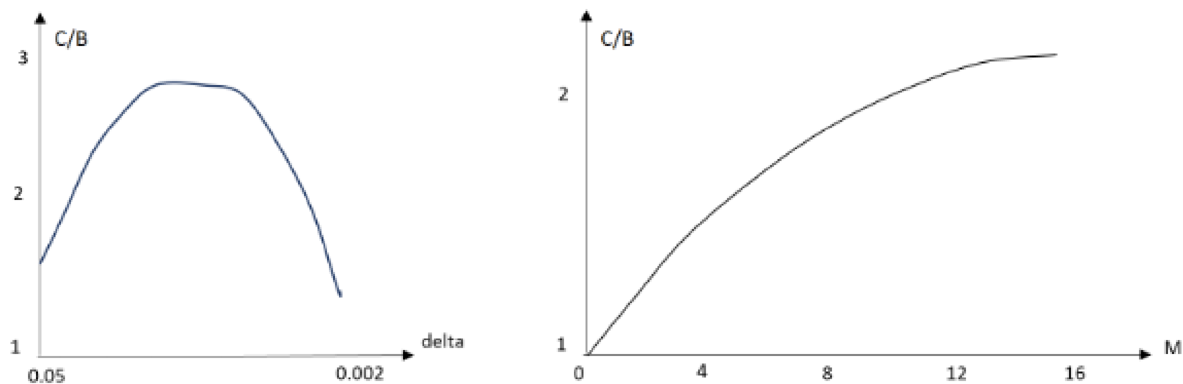



Figure 9: left: The acceleration factor as a function of delta; right: The acceleration factor as a function of M.

Table 1 is organized as follows: the first column lists the reference to the dataset from Figure 7, the columns labeled A, B, C contain the framerates obtained with the following optimization methods:

A. No optimization.

B. Two-pass adaptive screen sampling.

C. Two-pass adaptive screen sampling plus partitioning the interval of integration into M = 8 pieces.

The next two columns contain the acceleration factors achieved using, respectively, two-pass adaptive screen sampling and the proposed algorithm. The last column contains the acceleration factors achieved only by using the proposed algorithm.

You can note from the last column of the Table 1 that proposed method, by itself, increases FPS by a factor of 1.5 on the data sets used. This factor becomes smaller if a significant number of rays end early (for example, for the bones see Figure 7 in screenshots (b) and (d)). The reason for this is that most intervals of integration are short and are harder to partition into smaller ones.

The efficiency of the method also depends on how coherent the dataset is. The less coherent it is, the more pixels need to be processed on the second pass. This can happen if the value of delta is lowered. Thus, to a first-order approximation, the dependence on the data sets' coherence can be replaced with a dependence on delta. Figure 8 shows the pixels processed on the second pass in visualizing data set 3 with delta values 0.05, 0.01 and 0.002. (Blue pixels are those for which the integral had to be computed over one interval; white, over two; yellow, over three; and red, over more than three.)

Figure 9 (left) shows the acceleration factor (relative to standard adaptive undersampling) as a function of delta. It can be seen from the graph that the efficiency of our method is at its maximum for medium levels of coherence. The reason for the decrease in performance on low coherence is that volume integrals need to be calculated for more pixels (red pixels in Figure 8). The decrease in