Adaptive Sampling in Three Dimensions for Volume Rendering on GPUs

Category: Research

ABSTRACT

Direct volume rendering of large volumetric data sets on programmable graphics hardware is often limited by the amount of available graphics memory and the bandwidth from main memory to graphics memory. Therefore, several approaches to volume rendering from compact representations of volumetric data have been published that avoid most of the data transfer between main memory and the graphics programming unit (GPU) at the cost of additional data decompression by the GPU. To reduce this performance cost, adaptive sampling techniques were proposed; which are, however, usually restricted to the sampling in view direction.

In this work, we present a GPU-based volume rendering algorithm with adaptive sampling in all three spatial directions; i.e., not only in view direction but also in the two perpendicular directions of the image plane. This approach allows us to reduce the number of samples dramatically without compromising image quality; thus, it is particularly well suited for many compressed representations of volumetric data that require a computational expensive GPU-based sampling of data.

Keywords: Scientific visualisation, visualisation of large data sets, volume visualization, direct volume rendering

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1 INTRODUCTION

Programmable graphics hardware offers an enormous computational performance and is therefore particularly attractive for volume visualization—even on a single desktop PC. For this hardware configuration, memory and bandwidth limitations are usually the primary obstacles to real-time direct volume rendering of large volumetric meshes. As the bandwidth between main memory and the GPU is usually too small to allow for transferring large volumetric data sets for each frame at interactive rates, many GPU-based implementations are designed to store the volumetric data in graphics memory in order to avoid this bandwidth bottleneck.

While this approach allows for interactive volume rendering, it is limited to data sets that fit into graphics memory. To overcome this problem, several approaches to GPU-based volume rendering from compressed data have been published as programmable GPUs are flexible enough for implementations of complex data structures, which are well suited for more compact representations of volumetric data. Therefore, limited onboard memory can be traded for rapidly growing computational performance of programmable graphics hardware. While this approach avoids limitations of data transfer and of graphics memory, it requires considerably more computational performance of the GPU; on the one hand because larger volumetric grids are visualized, i.e., more sampling operations are required, and on the other hand because each sampling operation is computationally more expensive as it requires access to a complex data structure.

In order to improve the rendering performance, it is therefore crucial to reduce the number of sampling operations; in particular by adaptive sampling. However, the adaptive sampling techniques of published GPU-based volume renderers are usually restricted to varying the sampling rate on each view ray without adaptive sampling of the image plane. Although the latter is a well-known concept in ray tracing, the graphics pipelines of today's GPUs were not designed to support this approach; thus, it is not straightforward to accomplish an efficient GPU-based implementation of adaptive sampling of the image plane.

Our approach is based on an early depth test for adaptively reducing the number of sampling points in all three spatial dimensions and offers the additional advantage of biquadratic B-spline filtering of sampled data instead of bilinear interpolation of colors; thereby we avoid both, color blurring and bilinear interpolation artifacts.

After reviewing related work in Section 2, we discuss our algorithm and implementation in Section 3. Results are presented in Section 4. We conclude this paper in Section 5 with a discussion of future work.

2 RELATED WORK

Texture-based direct volume rendering was first published by Cabral et al. [2]. It exploits hardware support for three-dimensional textures by rasterizing a set of textured slices. Engel et al. [4] have exploited multi-texturing and dependent texture reads to implement pre-integrated volume rendering in a texture-based volume renderer.

Ray casting algorithms invert the order in which samples are processed by completely evaluating the volume rendering integral for each view ray before processing the next ray. While recently published GPU-based ray casters implemented this approach (e.g., the system by Stegmaier et al. [12]), earlier GPU-based implementations (e.g., by Roettger et al. [11] or Krüger and Westermann [6]) could only evaluate a small part of each integral in a GPU-based fragment program of limited length. The latter systems could be characterized as "ray marchers," which have to execute a fragment program multiple times by rasterizing polygons that cover the projection of the volume data; e.g., by rasterizing a screen-filling polygon or the front faces of a convex bounding volume.

Usually, a GPU-based ray marcher stores the current sampling point for each pixel and advances it along the corresponding view ray. Therefore, the order of sampling operations is similar to slice-based volume rendering. However, ray marchers can choose the starting point and the sampling distance independently for each pixel; thus, they are better suited for adaptive sampling of view rays as demonstrated by Roettger et al. [11]. Unfortunately, independently adapting the sampling distances on adjacent view rays will usually destroy the contiguous front of sampling points; thus, this approach is less appropriate for an adaptive sampling of slices.

Adaptive sampling on view rays is one of several well-known acceleration techniques for software volume ray casting [9]. Krüger and Westermann [6] have adapted empty-space skipping and early ray termination for programmable graphics hardware by exploiting an early "fragment-depth test." If a fragment program does not modify the fragment's depth, the depth test may be performed before the fragment program is executed. Thus, if the fragment's depth fails the depth test, it is not necessary to executed the fragment program for this particular fragment. We use the term "fragment-depth test" instead of "depth test" or "z test" to distinguish it from a "pixel-depth test" or specifically the "depth bounds test" [5], which compares user-defined bounds to the depth value already stored in the framebuffer's pixel at the location determined by the incoming fragment's coordinates. The pixel-depth test is independent of the depth of the incoming fragment; thus, it can be an early test even if the depth of the incoming fragment is computed by a fragment program. A pixel-depth test was employed by Neo-

phytou and Mueller [10] for empty-space skipping and early ray termination in GPU-based volume rendering using image aligned splatting.

Due to the limited amount of texture memory on graphics hardware, texture compression has been a research topic for many years. New features of programmable graphics hardware, in particular dependent texture reads, have led to additional efforts to implement random access to compact data representations in textures; a survey of many recent publications about GPU-based data structures is given by Lefohn et al. [7].

The main focus of these publications has been on random access to GPU-based implementations of basic data structures. There are considerably fewer publications about more efficient decoding techniques that are less efficient for random access but more efficient if many elements are accessed. One example is "deferred filtering" proposed by Lefohn et al. [8]. If this technique is extended to an adaptively sampled image plane, the size of the filter kernel has to be adapted correspondingly. Fortunately, pyramidal algorithms introduced by Burt [1] for image filtering can be employed for an appropriate sampling scheme as discussed in Section 3.2.

3 Adaptive Sampling in Three Dimensions

The goal of our approach is to adaptively choose the employed sampling distance according to an oracle that returns a preferable sampling distance d for any point in space. Our algorithm does not guarantee any actual sampling rates; instead, the sampling distance d returned by the oracle for a sampling point is used to "skip" sampling points that are closer than this distance.

For an efficient GPU-based implementation, the oracle has to be implementable in a fragment program. Since the oracle's input consists only of a three-dimensional point, the returned sampling distance has to be valid for all sampling directions. We do not place further restrictions on the oracle; thus, the oracle may be view-dependent or view-independent, it may be precomputed or computed on request, etc. However, it should be noted that our algorithm always samples both, the oracle and the volumetric data for the same points. This is often beneficial since the computation of the oracle might be considerably less expensive if performed together with a sampling operation of the visualized data for the same point.

We present our algorithm and GPU-based implementation in four steps: Section 3.1 discusses the skipping of sampling points for adaptively sampled view rays. The technique is similar to the skipping of sampling points with the help of an early fragment-depth test by Krüger and Westermann [6]. Similarly to the system by Roettger et al. [11] an oracle is employed to suggest a preferred sampling rate. The algorithm presented in Section 3.1 is actually less efficient than adapting the distance between sampling points as suggested by Roettger et al. [11]; however, it may be extended for locally adaptive sampling of each slice because sampling points are systematically aligned on slices. Section 3.2 discusses a variant of our GPU-based implementation [13] of the pyramid algorithm [1] for adaptive sampling and synthesizing an image that represents the data on the 0th slice. The extension of this algorithm for multiple slices and the actual volume integration with the help of pre-integrated transfer functions, is discussed in Section 3.3. This algorithm requires considerably less volumetric sampling operations than previously published volume rendering algorithms since it employs the oracle's results not only for an adaptive sampling of view rays but also for an adaptive sampling of slices. Our GPU-based implementation of the algorithm is discussed in Section 3.4.

3.1 Adaptively Skipping Sampling Points

To allow for an adaptive sampling of the image plane, we align all sampling points on view-aligned slices; see Figure 1. The slices start in front of the bounding volume and end behind the bounding volume. The uniform distance between slices corresponds to the distance between neighboring view rays. This allows us to specify all sampling distances in units of this smallest sampling distance. Samples outside the bounding volume are mapped to a default data value, which should be mapped to a transparent color for volume rendering.

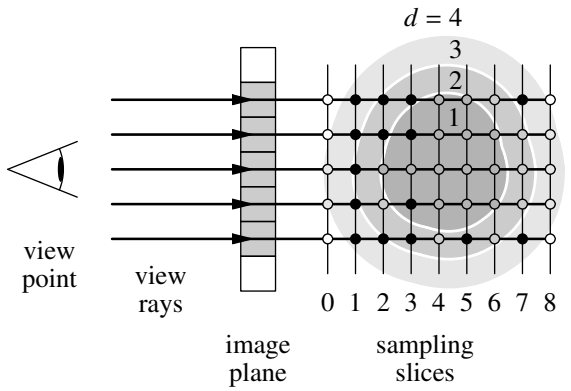


Figure 1: Adaptive sampling on view rays according to an oracle, which computes a sampling distance d (represented by gray levels) for each sampling point. Filled black dots represent skipped sampling points.

Table 1: Actual values of l_{slice} and $2^{l_{\text{slice}}+1}$ for i_{slice} from 0 to 8.

i_{slice}	0	1	2	3	4	5	6	7	8
l_{slice}	∞	0	1	0	2	0	1	0	3
$2^{l_{\text{slice}}+1}$	∞	2	4	2	8	2	4	2	16

Adaptive sampling is achieved by querying an oracle for a sampling distance d at each sampling point starting on the 0th slice, i.e., the frontmost slice. The result is stored together with the data sample for the same point in the frame buffer and updated with new values of each sampling point on the same view ray unless the sampling point is skipped. Sampling points on the next slices that are projected to the same pixel (i.e., that are on the same view ray) are skipped in dependency of the stored distance d; however, at most $d-1$ sampling points are skipped; thus, the sampling rate does not drop below the sampling rate required by the oracle if d is not less than 1 in the units described above. However, instead of skipping exactly $d-1$ sampling points, our algorithm skips sampling points until it reaches the next slice with an index that is divisible by an appropriate power of 2 in order to align sampling points of different rays on the same slice as required for the adaptive sampling of slices, which is discussed in Sections 3.2 and 3.3.

Figure 1 illustrates the skipping of sampling points. For the purpose of illustration, a view-independent oracle is assumed, which returns only integer sampling distances d from 1 to 4 represented by gray levels. Skipped sampling points are indicated by filled black dots while the color of all other sampling points represents the result of the oracle at each point.

The condition for skipping a sampling point on the i_{slice}-th slice is $d \geq 2^{l_{\text{slice}}+1}$ where d is the distance stored in the framebuffer and

$$l_{\text{slice}} = \max\{l \in \mathbb{N}_0 \mid i_{\text{slice}} \bmod 2^l = 0 \wedge l \leq l_{\max}\}$$

with a positive integer $l_{\max}$ that bounds the maximum actual sampling distance from above by $d = 2^{l_{\max}}$. The index of the last slice should be divisible by $2^{l_{\max}}$ to ensure sufficient sampling at boundaries. Note that l_{slice} is just the number of trailing zeros (bounded from above by $l_{\max}$) in a binary representation of the index i_{slice}.

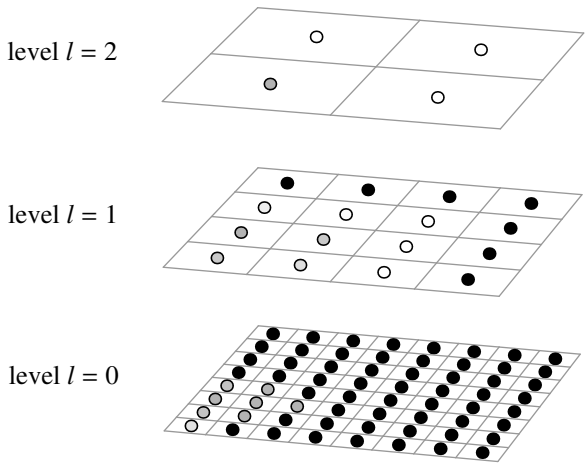


Figure 2: Adaptive sampling of a slice ($l_{\text{slice}} = l_{\max} = 2$). Gray levels represent sampling distances returned by an oracle. Filled black dots represent skipped sampling points.

Specific numbers for $l_{\max} = \infty$ are given in Table 1. One important feature of this scheme is that the skipping condition relies exclusively on previously stored results of the oracle and the index of the slice. This is crucial for an implementation that relies on early pixel-depth tests.

3.2 Adaptive Sampling of the Zeroth Slice

The adaptive skipping of sampling points on view rays discussed in the previous section reduces the number of sampling points quite efficiently; in particular, it is not restricted to empty space skipping but may also reduce the sampling rate if the data is locally bandwidth-limited. However, the number of samples can be further reduced by adaptively sampling each slice. If the adaptive sampling of a view ray results in a reduction by a certain factor, the combined effect of adaptively sampling in three dimensions reduces the number of samples by the third power of this factor. This section discusses the adaptive sampling of one slice (in fact, the 0th slice), while the combination with the adaptive sampling on view rays is described in the next Section 3.3.

The data required for the sampling of all view rays (of the finest level) is stored in an image of $w \times h$ pixels. For the adaptive sampling of this image on the 0th slice, our algorithm employs a pyramidal synthesis [1, 13] on $l_{\max} + 1$ levels. All pixels of the topmost level $l_{\max}$ are sampled while pixels on lower levels are either synthesized or also sampled if the oracle requires a smaller sampling distance. Figure 2 illustrates the adaptive sampling of the 0th slice for $l_{\max} = 2$. Pseudo code for the adaptive sampling of any slice is given in Figure 3. For the 0th slice, the relevant function call is PROJECTSLICE(0); i.e., i_{slice} is equal to 0. In this case l_{slice} is set to $l_{\max}$.

The pyramidal synthesis for the 0th slice starts from the coarsest level $l_{\max}$ of size $w/2^{l_{\max}} \times h/2^{l_{\max}}$ (assuming that w and h are divisible by $2^{l_{\max}}$). On this topmost level $l_{\max}$ no sampling points may be skipped; therefore, data values $s_{l_{\max}}[x,y]$ are sampled from the volumetric data and sampling distances $d_{l_{\max}}[x,y]$ are requested from the oracle for all pixels (x,y) of level $l_{\max}$. The coordinates of the sampling point for a pixel (x,y) on level l is determined by a mapping to pixel coordinates on level 0 and thereby to a sampling point on the corresponding view ray. The subroutine SAMPLE(l,x,y) samples the volumetric data at the three-dimensional point corresponding to

```
PROJECTSLICE(i_slice):
  l_slice ← max{l ∈ ℕ0 | i_slice mod 2^l = 0 ∧ l ≤ l_max}
  for l ← l_slice to 0 do
    for x ← 0 to w/2^l − 1 do
      for y ← 0 to h/2^l − 1 do
        if l < l_slice and (l_slice = l_max
        or d_l[x,y] < 2^(l_slice+1)) then
          d_l[x,y] ← SYNTHESIZE(d,l,x,y)
          s_l[x,y] ← SYNTHESIZE(s,l,x,y)
        endif
        if (l = l_max or d_l[x,y] < 2^(l+1)) then
          d_l[x,y] ← ORACLE(l,x,y)
          s_l[x,y] ← SAMPLE(l,x,y)
        endif
        if l = 0 and (l_slice = l_max
        or d_l[x,y] < 2^(l_slice+1)) then
          ACCUMULATECOLOR(x,y,i_slice)
        endif
      endfor y
    endfor x
  endfor l
```

Figure 3: Pseudo code for adaptive sampling of the i_{slice}-th slice. See Figure 5 for the specification of SYNTHESIZE, SAMPLE, and ORACLE; ACCUMULATECOLOR is specified in Figure 6..

the pixel (x,y) of level l of the current slice, while ORACLE(l,x,y) returns the sampling distance for this point determined by an oracle. Pseudo code for these subroutines is presented in Figure 5.

For each pixel (x,y) of the next level $l = l_{\max} - 1$, the algorithm synthesizes (i.e., approximates) scalar data $s_l[x,y]$ and a sampling distance $d_l[x,y]$ from the values on level $l+1 = l_{\max}$. Since the actual distance between the sampling points corresponding to two adjacent pixels of level l is 2^l, sampling on this level l is necessary if the synthesized sampling distance suggested by the oracle is less than 2^{l+1}. In this case the volumetric data is sampled and the oracle's sampling distance is computed and stored in $s_l[x,y]$ and $d_l[x,y]$, respectively.

The synthesis of data and distances on finer levels is specified by the function SYNTHESIZE(s,l,x,y) and SYNTHESIZE(d,l,x,y), respectively. An example is presented in Figure 4. Unfortunately, the formal specification in Figure 5 of this function obscures the actually quite simple scheme. The scheme corresponds to the regular Doo-Sabin subdivision scheme and the subdivision of biquadratic B-splines [3].

This synthesis scheme is particularly well suited for a GPU-based implementation since it can be implemented by the bilinear interpolation of a single texture lookup with appropriate texture coordinates as indicated by the gray dot in Figure 4. More details and applications to problems in image processing are discussed by Strengert et al. [13].

3.3 Putting It Together

The pseudo code in Figure 3 already covers the adaptive sampling of all slices. The main difference between the sampling of the 0th slice discussed in Section 3.2 and the sampling of further slices is the computation of l_{slice}. While l_{slice} is equal to $l_{\max}$ for the 0th slice, it is computed as discussed in Section 3.1 for a general slice index i_{slice}.

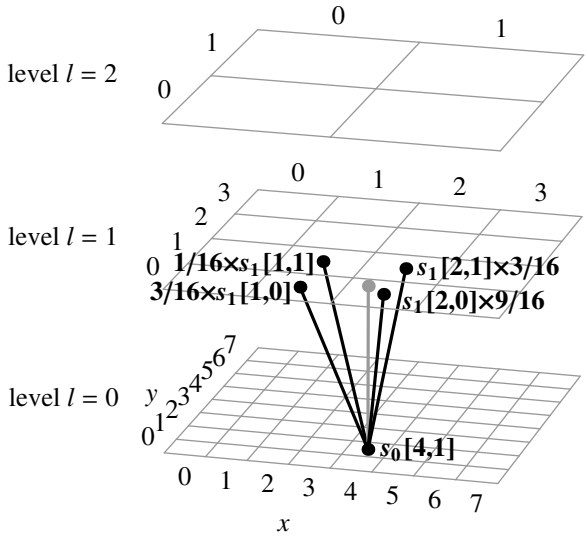


Figure 4: Synthesis of $s_0[4,1] = \frac{3}{16}s_1[1,0] + \frac{9}{16}s_1[2,0] + \frac{1}{16}s_1[1,1] + \frac{3}{16}s_1[2,1]$. The gray dot indicates the texture coordinates for the corresponding bilinear texture lookup.

SYNTHESIZE(g,l,x,y): *¡g is either s or d!*

return

$$\left(\tfrac{1}{4}+\left(\tfrac{x}{2}-\lfloor\tfrac{x}{2}\rfloor\right)\right)\left(\tfrac{1}{4}+\left(\tfrac{y}{2}-\lfloor\tfrac{y}{2}\rfloor\right)\right)g_{l+1}\left(\lceil\tfrac{x}{2}\rceil-1,\lceil\tfrac{y}{2}\rceil-1\right)$$
$$+\left(\tfrac{3}{4}-\left(\tfrac{x}{2}-\lfloor\tfrac{x}{2}\rfloor\right)\right)\left(\tfrac{1}{4}+\left(\tfrac{y}{2}-\lfloor\tfrac{y}{2}\rfloor\right)\right)g_{l+1}\left(\lceil\tfrac{x}{2}\rceil,\lceil\tfrac{y}{2}\rceil-1\right)$$
$$+\left(\tfrac{1}{4}+\left(\tfrac{x}{2}-\lfloor\tfrac{x}{2}\rfloor\right)\right)\left(\tfrac{3}{4}-\left(\tfrac{y}{2}-\lfloor\tfrac{y}{2}\rfloor\right)\right)g_{l+1}\left(\lceil\tfrac{x}{2}\rceil-1,\lceil\tfrac{y}{2}\rceil\right)$$
$$+\left(\tfrac{3}{4}-\left(\tfrac{x}{2}-\lfloor\tfrac{x}{2}\rfloor\right)\right)\left(\tfrac{3}{4}-\left(\tfrac{y}{2}-\lfloor\tfrac{y}{2}\rfloor\right)\right)g_{l+1}\left(\lceil\tfrac{x}{2}\rceil,\lceil\tfrac{y}{2}\rceil\right)$$

SAMPLE(l,x,y):

return sampled scalar data at the position corresponding to pixel $\left(2^l\left(x+\frac{1}{2}\right)-\frac{1}{2}, 2^l\left(y+\frac{1}{2}\right)-\frac{1}{2}\right)$ of level 0.

ORACLE(l,x,y):

return sampled oracle at the position corresponding to pixel $\left(2^l\left(x+\frac{1}{2}\right)-\frac{1}{2}, 2^l\left(y+\frac{1}{2}\right)-\frac{1}{2}\right)$ of level 0.

Figure 5: Pseudo code for synthesizing data and sampling scalar data and the oracle. The functions are employed in Figure 3.

l_{slice} determines the starting level for the loop over l in the pyramidal synthesis. For example, the 0th slice processes all levels, while the 1st slice processes only the bottommost level 0. After the first sampling on the 0th slice, level 1 is updated for the first time on the 2nd slice and level 2 on the 4th slice, etc. In general, samples on level l are updated for the first time on the 2^l-th slice. More samples for level l are unnecessary since the distance between sampling points of adjacent pixels on level l is 2^l in units of the minimum distance between sampling points. Note that data that has been synthesized from data of level l, is always replaced by new samples on level $l-1$ taken at smaller sampling distances if required by the oracle.

The condition $d < 2^{l_{\text{slice}}+1}$ for actually performing the sampling (see Section 3.1) becomes $l < l_{\text{slice}}$ **and** $d_l[x,y] < 2^{l_{\text{slice}}+1}$ in Figure 3. The first part $l < l_{\text{slice}}$ ensures that we do not synthesize new values from data of level $l_{\text{slice}}+1$, which is not updated for this slice since the loop over l starts with l_{slice}. The remaining algorithm for sampling slices is exactly as in the case of the 0th slice, which was discussed in Section 3.2.

ACCUMULATECOLOR(x,y,i_{slice}):

if $i_{\text{slice}} > 0$ **then**

$C[x,y] \leftarrow C[x,y] + (1-\alpha[x,y]) \times$
$C_{\text{lut}}\left(s_{\text{prev}}[x,y], s_0[x,y], i_{\text{slice}} - i_{\text{prev}}[x,y]\right)$

$\alpha[x,y] \leftarrow \alpha[x,y] + (1-\alpha[x,y]) \times$
$\alpha_{\text{lut}}\left(s_{\text{prev}}[x,y], s_0[x,y], i_{\text{slice}} - i_{\text{prev}}[x,y]\right)$

endif

$s_{\text{prev}}[x,y] \leftarrow s_0[x,y]$

$i_{\text{prev}}[x,y] \leftarrow i_{\text{slice}}[x,y]$

PROJECTVOLUME(n_{slices}):

for $x \leftarrow 0$ **to** $w-1$ **do**

for $y \leftarrow 0$ **to** $h-1$ **do**

$C[x,y] \leftarrow 0$

$\alpha[x,y] \leftarrow 0$

endfor y

endfor x

$i_{\max} \leftarrow 2^{l_{\max}} \lceil n_{\text{slices}} / 2^{l_{\max}} \rceil$

for $i_{\text{slice}} \leftarrow 0$ **to** $i_{\max}$ **do**

PROJECTSLICE(i_{slice})

endfor i_{slice}

Figure 6: Pseudo code for the color accumulation employed in Figure 3 and the main loop over all slices for volume rendering.

Pseudo code for the actual volume rendering is presented in Figure 6. The main function PROJECTVOLUME(n_{slices}) initializes arrays of size $w \times h$ for accumulated, opacity-weighted colors $C[x,y]$ and opacities $\alpha[x,y]$ to transparent black. As mentioned in Section 3.1, the index of the last slice has to be divisible by $2^{l_{\max}}$; thus, the maximum slice index $i_{\max}$ is increased accordingly and the volume rendering is performed in a loop over all slices; see Figure 3 for the adaptive sampling specified by the function PROJECTSLICE(i_{slice}).

The volume rendering integral for each pixel on the finest level 0 is approximated by a front-to-back color accumulation performed by the method ACCUMULATECOLOR(x,y,i_{slice}), which is also defined in Figure 6. For all but the 0th slice, colors $C[x,y]$ and opacities $\alpha[x,y]$ are determined by a lookup in a pre-integrated table for opacity-weighted colors $C_{\text{lut}}(s_{\text{front}}, s_{\text{back}}, d)$ and opacities $\alpha_{\text{lut}}(s_{\text{front}}, s_{\text{back}}, d)$ for the color and opacity of a ray segment of length d with data samples s_{front} and s_{back} at its ends. After colors are blended, the end point of the current segment and the data sample for this point is stored as the starting point of the next segment.

The presented algorithm was designed for an implementation on GPUs supporting certain features as discussed in the next section.

3.4 Putting It Onto a GPU

For a GPU-based implementation of the algorithm presented in Figures 3, and 6, we pack the arrays $s_{\text{prev}}[x,y]$, $i_{\text{slice}}[x,y]$, $s_l[x,y]$ and $d_l[x,y]$ for all levels l from 0 to $l_{\max}$ in one 16-bits floating-point texture image of size $\frac{3}{2}w \times h$ with RGBA components. In a second 16-bits floating-point texture image colors $C[x,y]$ and $\alpha[x,y]$ are accumulated. While these texture images are also used as renderbuffer images, additional read-only texture images are required for the pre-integration table and the volumetric data.

Our implementation is based on a texture-based volume renderer with view-aligned slices; thus, we can easily implement the loop over all pixels (x,y) in Figure 3 by rasterizing slice polygons. The loop over l is implemented by rasterizing several primitives into

```
PROJECTSLICE(i_slice):
  l_slice ← max{l ∈ ℕ0 | i_slice mod 2^l = 0 ∧ l ≤ l_max}
  for l ← l_slice to 0 do
    for x ← 0 to w/2^l − 1 do
      for y ← 0 to h/2^l − 1 do
        if l < l_slice then
          ¡synthesize all pixels!
          d_l[x,y] ← SYNTHESIZE(d,l,x,y)
          s_l[x,y] ← SYNTHESIZE(s,l,x,y)
        endif
        if d_l[x,y] < 2^(l+1) then
          ¡sample and accumulate!
          z_l[x,y] ← (4/6 + i_max − i_slice)
        else if d_l[x,y] < 2^(l_slice+1) then
          ¡only accumulate!
          z_l[x,y] ← (2/6 + i_max − i_slice)
        else  ¡neither sample nor accumulate!
          z_l[x,y] ← (0/6 + i_max − i_slice)
        endif
        if (l = l_max or
        (3/6 + i_max − i_slice) < z_l[x,y]) then
          d_l[x,y] ← ORACLE(l,x,y)
          s_l[x,y] ← SAMPLE(l,x,y)
        endif
        if l = 0 and (l_slice = l_max
        or (1/6 + i_max − i_slice) < z_l[x,y]) then
          ACCUMULATECOLOR(x,y,i_slice)
        endif
      endfor y
    endfor x
  endfor l
```

Figure 7: Pseudo code for a variant of the algorithm presented in Figure 3, which is suitable for an implementation based on an early fragment-depth test.

different parts of the texture image according to the packing of the image pyramid into the texture image. Thus, for each rasterized primitive, l, l_{slice}, and l_{max} are constant. The block of the innermost loop in Figure 3 consists of three "if-endif"-blocks as indicated by horizontal lines. We implement these three "if-endif"-blocks by conditionally rasterizing the same primitive up to three times with different fragment programs.

Since we store all levels of the image pyramid in one texture image, we have to use the same image as texture image and as renderbuffer image. Simultaneously rendering into and reading from an image is not a well specified operation; however, sending a "fence" [5] without waiting for it after each primitive apparently avoids all caching problems at an acceptable performance cost in our implementation. Alternatively, a more complicate ping-pong rendering scheme could be employed that switches the roles of two texture/renderbuffer images after each primitive; see Strengert et al. [13] for more details.

Those terms of the "if"-conditions in the pseudo code presented in Figure 3 that depend only on per-primitive constants (l, l_{slice}, and l_{max}) are evaluated on the CPU and the remaining terms that depend on $d_l[x,y]$ can be implemented by a potentially early pixel-depth test. Unfortunately, we were not able to implement this algorithm without deactivating the early depth bounds test of our target hardware. Thus, we had to revise our algorithm for an implementation based on an early fragment-depth test; i.e., a standard OpenGL depth test with comparison relation "less" on our target hardware. This test can conditionally avoid the execution of the fragment program that samples the volumetric data. In the revised pseudo code, which is presented in Figure 7, the depth buffer required for the fragment-depth test is denoted by $z_l[x,y]$. Results obtained with our implementation of this algorithm are reported in Section 4.

The only disadvantage of this variant are the additional synthesis operations, which are not avoided by early depth tests. However, the potentially extremely expensive sampling of the volumetric data is successfully avoided by early fragment-depth tests while the synthesis of data is implemented by a single bilinear interpolated texture lookup regardless of the representation of the volumetric data.

Table 2: Results for the aneurysm data set on a 512×512 viewport.

algorithm / distance oracle	tex.-based min.	adaptive sampling min.	adaptive sampling $\lvert\nabla\alpha\rvert > \varepsilon$
# fragments	49 M	162 M	61 M
# sampling ops	99 M	56 M	1 M
time in secs	0.87	0.79	0.23

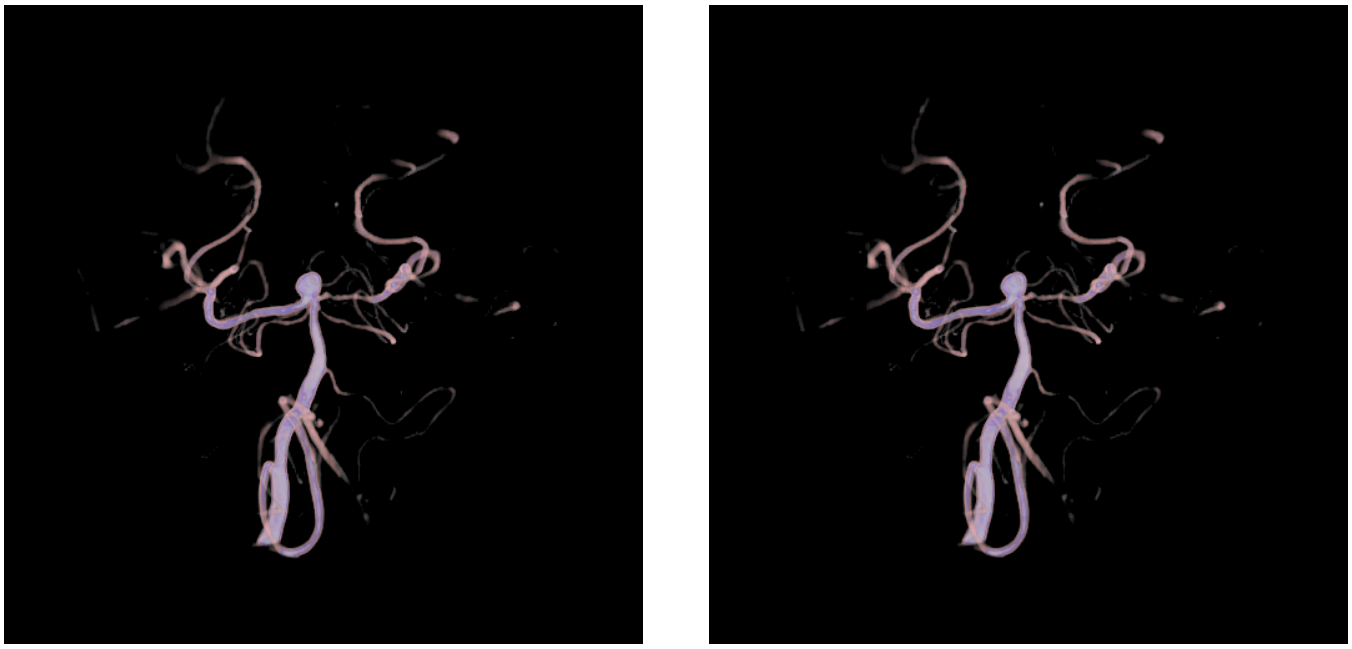
Table 3: Results for the abdomen data set on a 512×512 viewport.

algorithm / distance oracle	tex.-based min.	adaptive sampling min.	adaptive sampling $\lvert\nabla\alpha\rvert > \varepsilon$
# fragments	37 M	122 M	61 M
# sampling ops	74 M	42 M	9 M
time in secs	0.76	0.68	0.38

4 RESULTS

We tested the implementation described in Section 3.4 on a Linux-PC equipped with an Intel Pentium 4 processor (3.4 GHz) and an NVIDIA 7800 GTX 512 graphics adapter with 512 MB of video memory. For comparison with a pre-integrated texture-based volume rendering system we employed cartesian data sets that are stored in three-dimensional texture maps. Therefore, the sampling of the volume data is performed by a simple hardware-accelerated trilinear texture lookup. The time required for this 3D texture sampling is far less than the time required to access complex, GPU-based data structures (see for example Lefohn et al. [7]); thus, the timings of our current implementation are not representative for actual applications. However, the timings prove that many of the potentially costly sampling operations are in fact avoided by early depth tests.

Table 2 summarizes our results for a $512 \times 512 \times 512 \times 8$ bits data set of an aneurysm (courtesy of Michael Meißner, Viatronix Inc., USA). For comparison, the first column shows measurements for a pre-integrated, texture-based volume renderer with the same number of slices. Figure 8a shows the volume rendering. The second column presents statistics for our algorithm with an oracle that always returns the minimum sampling distance; i.e., the rendering is at full resolution. The resulting visualization is visually indistinguishable from Figure 8a and was, therefore, not included. The results for this oracle were included to present the worst-case overhead of our algorithm. The third column shows our algorithm with adaptive sampling according to an ad-hoc oracle that is pre-computed and stored with the data set in the same



(a) (b)

Figure 8: Volume rendering of the aneurysm data set: (a) Texture-based volume rendering for comparison. (b) Our adaptive sampling according to a pre-computed ad-hoc oracle.

three-dimensional texture map. Roughly speaking, it estimates the distance to the nearest voxel that features a gradient magnitude of the opacity greater than some threshold. The resulting visualization is presented in Figure 8b and achieves almost the same quality as the reference rendering with considerably fewer sampling operations. $l_{\max}$ was set to 3 in our examples; i.e., the maximum sampling distance is 8.

The first row of data lists the number of fragments generated during the rendering. Since our algorithm has to rasterize up to three primitives for each level, it generates considerably more fragments. The second row shows the number of required sampling operations of the volumetric data (i.e., texture lookups), which is twice the number of fragments for the texture-based volume renderer because of the pre-integration but includes only the sampling operations that pass the depth test for our adaptive sampling system. Note that not all of the rejected fragments are rejected early; however, the timings indicate that many of them are rejected before the sampling operation was performed. The last row specifies timings in seconds.

Our second example is a resampled $512 \times 512 \times 512 \times 8$ bits version of an abdomen data set (the original data set is also courtesy of Michael Meißner). The pre-integrated texture-based volume rendering is shown in Figure 9a, while the adaptively sampled rendering is presented in Figure 9b. Measurements are summarized in Table 3. Since this data set contains fewer empty regions, our adaptive sampling algorithm requires more sampling operations than for the aneurysm data set but still significantly less than the texture-based volume renderer.

5 Conclusions and Future Work

We implemented an efficient GPU-based volume rendering algorithm on current graphics hardware with adaptive sampling in view direction and in the two perpendicular directions of the image plane. The adaptive sampling rate is determined by an almost arbitrary oracle. Our measurements show that we can avoid expensive sampling operations of complex volumetric data structures with the help of an early fragment-depth test. Thus, our main contribution is a GPU-friendly algorithm (Sections 3.3 and 3.4) that combines adaptive sampling in view direction (Section 3.1) with a pyramidal synthesis for adaptive sampling of slice images (Section 3.2).

Adaptive sampling according to an oracle in general, and our algorithm in particular, is especially beneficial if the sampling operation is computational expensive; for example, for random access to compressed volumetric data and/or if the volume data features a complex boundary, and/or if there is a strong local variation of the required sampling rate, and/or if the sampling quality should be continuously adaptable, and/or if a high-resolution display results in excessive oversampling, etc. In the latter case of excessive oversampling, our current implementation avoids superfluous sampling; however, it requires additional synthesis operations. To reduce this effect efficiently, we can skip all slices with indices $i_{\text{slice}} \bmod 2^l \neq 0$ with an appropriate power l.

Future work includes improvements of our volume rendering algorithm regarding boundaries of the data and of the view frustum, the use of shells instead of slices for perspective projection, early ray termination, and applications to compressed GPU-based volumetric data structures and the design of appropriate oracles.

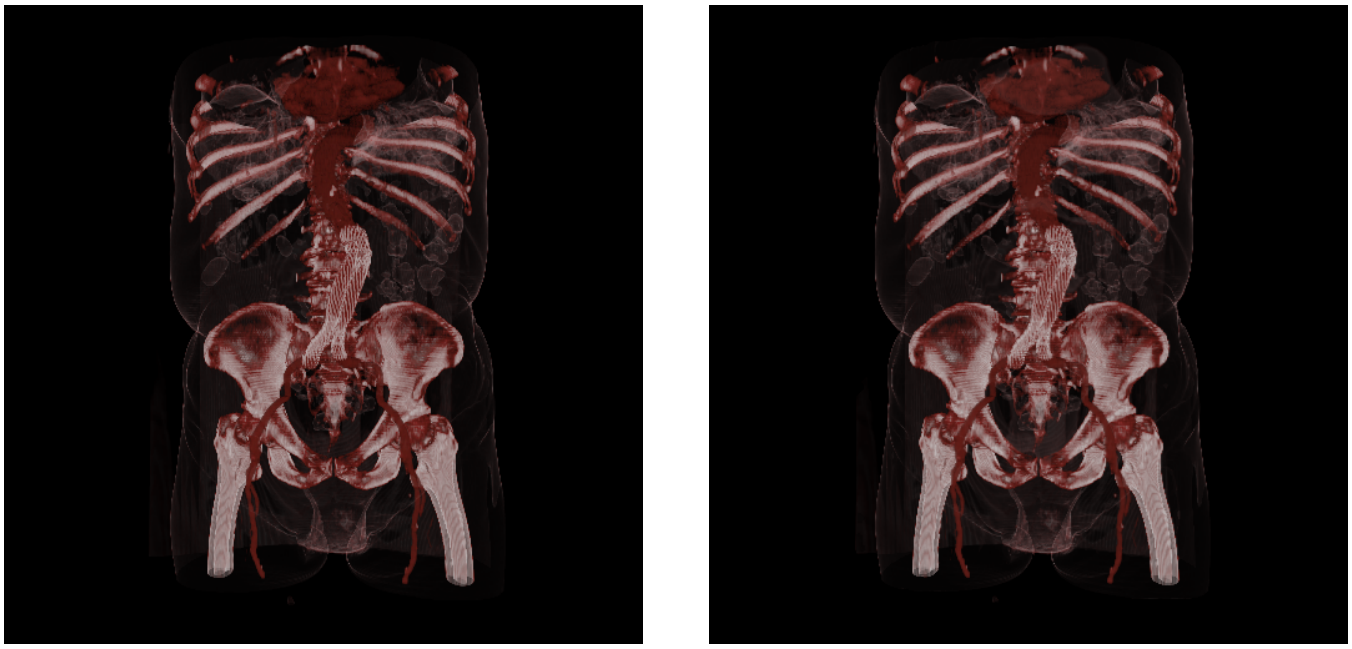
References

[1] P. J. Burt. Fast filter transforms for image processing. *Computer Graphics and Image Processing*, 16:20–51, 1981.

[2] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings 1994 Symposium on Volume Visualization*, pages 91–98, 1994.

[3] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10(6):350–355, 1978.

[4] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In W. Mark and A. Schilling, editors, *Proceedings Graphics Hardware 2001*, pages 9–16. ACM Press, 2001.



(a) (b)

Figure 9: Abdomen data set: (a) texture-based and (b) with our adaptive sampling algorithm.

[5] M. J. Kilgard. *NVIDIA OpenGL Extension Specifications*. NVIDIA Corporation, 2004.

[6] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[7] A. E. Lefohn, J. Kniss, R. Strzodka, S. Sengupta, and J. D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, 25(1), 2006.

[8] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker. A streaming narrow-band algorithm: Interactive deformation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):422–433, 2004.

[9] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[10] N. Neophytou and K. Mueller. Gpu accelerated image aligned splatting. In *Proceedings Volume Graphics 2005*, pages 197–205, 2005.

[11] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Proceedings Joint Eurographics-IEEE TCVG Symposium on Visualization (VisSym '03)*, pages 231–238, 2003.

[12] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware–based raycasting. In *Proceedings International Workshop on Volume Graphics '05*, pages 187–195, 2005.

[13] M. Strengert, M. Kraus, and T. Ertl. Pyramid methods in gpu-based image processing. accepted for publication at VMV 2006.