Multimedia Tools and Applications (2021) 80:20971–20989
https://doi.org/10.1007/s11042-021-10642-4

Acceleration techniques for cubic interpolation MIP volume rendering

Yongha Shin[1] · Bong-Soo Sohn[2] · Heewon Kye[3]

Received: 18 February 2020 / Revised: 8 December 2020 / Accepted: 4 February 2021 /
Published online: 12 March 2021

Abstract
Maximum intensity projection (MIP) is a volume visualization technique that is important in modern medical imaging systems. We propose a method to accelerate high-quality MIP volume rendering using cubic interpolation. First, our method skips more regions of volume data that do not affect the output image. To do this, we propose a method of transforming the B-spline interpolation function into a sub-division of Bezier spline interpolation. We generate the B-spline interpolation control points then the Bezier interpolation control points from three dimensional voxel values. The maximum value of each block is approximated using the Bezier interpolation control points due to the convex hull property of the Bezier spline. By accurately approximating the maximum value of each block, we can skip more unnecessary blocks. Second, we propose an efficient method of parallelization when performing volume visualization using a GPU. In order to reduce the number of memory transfers, our method determines the working shape of a warp, a bundle of 32 GPU threads, depending on the viewing direction. As a result, our method achieves a remarkable rendering speed improvement with no loss of image quality compared to previous studies, and performs high-quality MIP volume rendering using cubic interpolation at interactive speed.

Keywords Volume rendering · Cubic interpolation · Bezier spline · GPU memory divergence · Maximum intensity projection

✉ Heewon Kye
kuei@hansung.ac.kr

[1] Department of Information System Engineering, Hansung University, 116 Samseongyo-ro 16-gil, Seongbuk-gu, Seoul 02876, South Korea

[2] School of Computer Science and Engineering, Chung-Ang University, 84 Heukseok-ro, Dongjak-gu, Seoul 06974, South Korea

[3] Division of Computer Engineering, Hansung University, 116 Samseongyo-ro 16-gil, Seongbuk-gu, Seoul 02876, South Korea

1 Introduction

Volume rendering is a technique for generating images using volume data, which is composed of the three-dimensional array of voxels [8], that can be obtained when a patient has a CT scan or MRI taken. Modern medical imaging systems rely on this technique due to the difficultly in diagnosing hundreds of human body cross-sectional images taken by a CT scan or MRI one by one. Ray casting is currently the most popular method of volume rendering. In ray casting, rays proceed from the viewer through the projection plane (i.e. image) and extend into the target volume. Along the ray's path, intensity values are reconstructed at sampling positions in the volume coordinates [11] (Fig. 1). Reconstructed values are blended to generate the output pixel value. Alpha blending, maximum value selection, and averaging are the most popular methods of blending.

Maximum intensity projection (MIP) is a volume rendering technique that displays the maximum value along the viewing direction [14, 18] (Fig. 1). This technique is mainly used to observe blood vessels and skeletal structures [15].

To create the image, we need to find the continuous interpolation function using a given set of voxels [11]. There are many possible interpolation functions (or filter kernels). As described in previous research [4], tri-linear interpolation is a well-known method and generates smoother results than the nearest neighbor interpolation which generates staircase artifacts [15].

Many modern medical imaging systems require very high-quality images, where interpolation methods using a cubic spline are particularly useful [11]. When viewing enlarged images of the vascular structure, as shown in Fig. 2, images using cubic interpolation are clearly superior to those using linear interpolation. There are various ways to determine the coefficients of a cubic function, the choice of coefficients presents trade-offs between the sensitivity to noise and blurring. For example, the B-spline very smooth, but has low post-aliasing (pixel energy leaking), while the Catmull-Rom spline produces less smoothing but has poor post-aliasing properties, and there is no optimal setting that works for all applications [10]. On the other hand, the windowed sinc filter generates a very good image, but it is difficult to use practically for volume visualization because the amount of computation is excessive.

Producing such a high-quality image requires a significant amount of time, due to the increase in computation and memory references. For example, linear interpolation can be computed as a weighted sum of $2^3 = 8$ voxel values, but a cubic interpolation is calculated as a

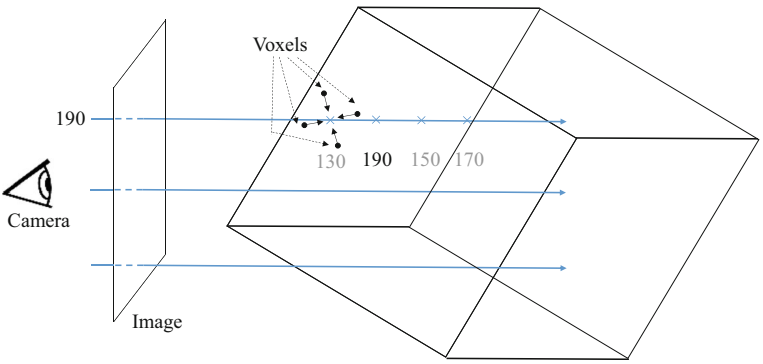
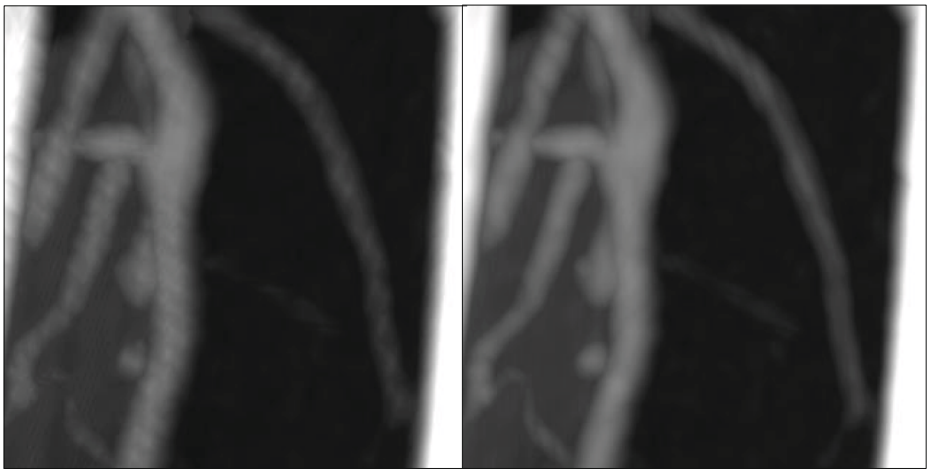


Fig. 1 Maximum intensity projection using the ray casting method, where the maximum value along the line of sight is determined



(a) (b)

Fig. 2 Magnified CT images using **a** linear interpolation and **b** cubic interpolation. The images that were processed with cubic interpolation were higher quality than those processed with linear interpolation when magnified

weighted sum of $4^3 = 64$ voxel values in a three-dimensional space. Even when utilizing GPUs [3], volume visualization using cubic interpolation takes a significant amount of time to generate high-resolution images. In this paper, we propose a more efficient MIP volume rendering method using cubic interpolation processed by GPUs.

First, we can skip unnecessary areas of volume data, with a 1.5 times better than the existing technique [25]. The maximum value of each area must be accurately estimated in order to determine whether or not it is necessary; a region with a low maximum value is more likely to be removed. To obtain more accurate estimates, we propose a new method that transforms the B-spline into piecewise Bezier splines and then subdivides each Bezier spline into smaller subsections.

Second, we suggest a new method to improve the computational efficiency of the GPU. When the viewing direction changes, we dynamically determine the optimum shape of tiles (group of pixels). We enhance memory access efficiency by utilizing the memory coalescing capability of the GPU [24].

The remainder of this paper is organized as follows. After reviewing the preliminaries and related work in Section 2, we describe our method for efficient space skipping using Bezier curve subdivision and GPU utilization in Section 3 and 4, respectively. Section 5 presents the experimental results. Finally, we make a conclusion in Section 6.

2 Related works

2.1 Empty space skipping

Sub-volumes that are not represented in the output image can be removed from the volume data to improve rendering time. Although it is not the subject of this paper, in direct volume rendering, users typically determine the transparent and opaque sections using the transfer

function, allowing the transparent sections to be removed in advance [9]. For the concrete method, the entire volume is subdivided into equal sized blocks, and the maximum and minimum values of every block (*M, m* in Fig. 3(a)) are calculated and stored during in the preprocessing time. During rendering, if the maximum and minimum values of a block are within in the transparency range designated by the user (*T[M,m]* in Fig. 3(a)), the block is determined to be transparent.

Because there is no transparent part in MIP rendering, an alternative method is used. As the ray progresses, if the current cumulative value is greater than the maximum value of the current block (*M* in Fig. 3(b)), the block is deemed unnecessary and does not affect the result [5, 13]. The processing of one pixel (or ray) is described in Algorithm 1.

Algorithm 1. MIP ray casting using empty space skipping

```
Processing one ray
pixel_max := 0
sample_position := start position of the ray
WHILE sample_position is in the volume
    block := GetBlockFrom (sample_position)
    IF block.max < pixel_max THEN skip block
    ELSE pixel_max := max ( ManySamplingsAndGetMaxFrom(block), pixel_max )
    sample_position := move to next block
```

2.2 B-spline interpolation

B-spline is a popular technique to generate curves using given control points. The curve is drawn as the weighted sum of the points according to the changing parameter *t*. In this study, we use the cubic B-spline using four control points (see Eq. 1). For example, the *i*-th curve, S_i, uses points P_{i-1}, P_i, P_{i+1}, and P_{i+2}, while the $(i+1)$-th curve uses points P_i, P_{i+1}, P_{i+2}, and P_{i+3}. Adjacent curves, S_i and S_{i+1}, are smoothly connected at the end points.

The B-spline does not pass through the given control points because $S_i(0) = \frac{1p_{i-1}+4p_i+1p_{i+1}}{6} \neq p_i$ for each S_i. At each voxel position (x, y, z) in the volume data, the B-spline reconstructed sampling value is different from the value measured by CT scan or MRI, i.e. $b(x, y, z) \neq volume[x][y][z]$. Therefore, B-spline is not an interpolation but an approximation method, and involves an over-blur problem.

$$S_i(t) = \begin{bmatrix} t^3 & t^2 & t^1 & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix} \quad (1)$$

To solve this problem, the control points are moved, such that the B-spline curve based on the moved control points passes through the original control points. This method is called B-spline interpolation [23]. As shown in Fig. 4, the control points, P_i, were moved to the control points, B_i, and the B-spline generated using points B_i then passes through the original points P_i.

We are able to obtain the control points B using the given control points P. The values for B[k] must be calculated such that Eq. 2 is satisfied for all integers *k*, while β is a B-spline basis of degree 3.

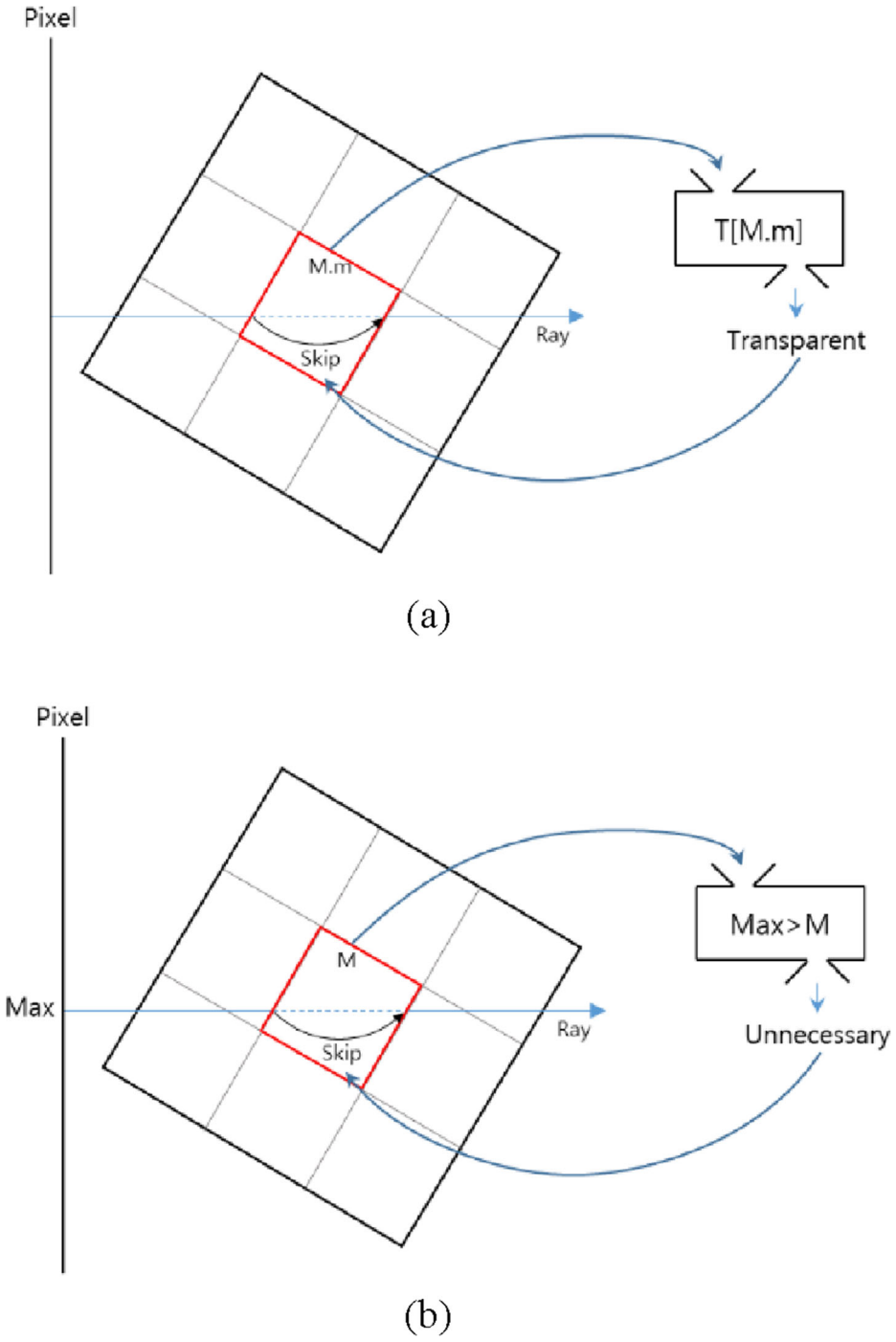


Fig. 3 Skipping an unnecessary block for direct volume rendering and MIP. **a** In case of direct volume rendering, the block is skipped using the user defined transfer function and the maximum and minimum values ($T[M, m]$) of the block (**b**) In the case of MIP, the block is skipped if the current cumulative value (Max) is greater than the block maximum value (M)

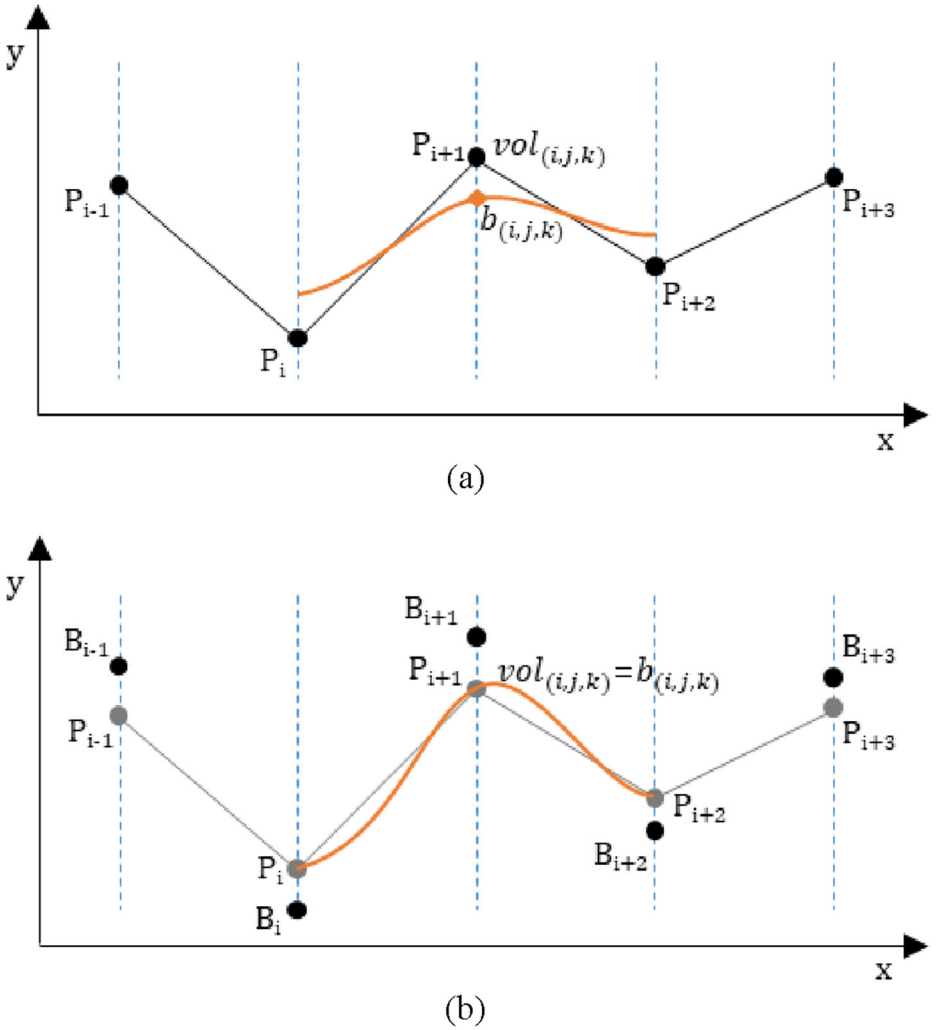


Fig. 4 **a** The B-spline does not pass through the given control points P_i **b** After the control points P_i are moved to the control points B_i, the B-spline then passes through the original points P_i

$$P[k] = \sum_{k' \in Z} B[k']\beta(k-k'), \beta(x) = \begin{Bmatrix} 0, & 2 \leq |x| \\ \frac{1}{6}(2-|x|)^3, & 1 \leq |x| < 2 \\ \frac{2}{3} - \frac{1}{2}|x|^2(2-|x|), & |x| < 1 \end{Bmatrix} \quad (2)$$

Essentially, we compute B to satisfy the expression $\frac{1B_{i-1}+4B_i+1B_{i+1}}{6} = P_i$ for each i. This problem can be solved using matrix formulations [2] which is a banded Toeplitz matrix. In this paper we utilize Ruijters's method [17] because of its simplicity, while other efficient methods exist [1, 16].

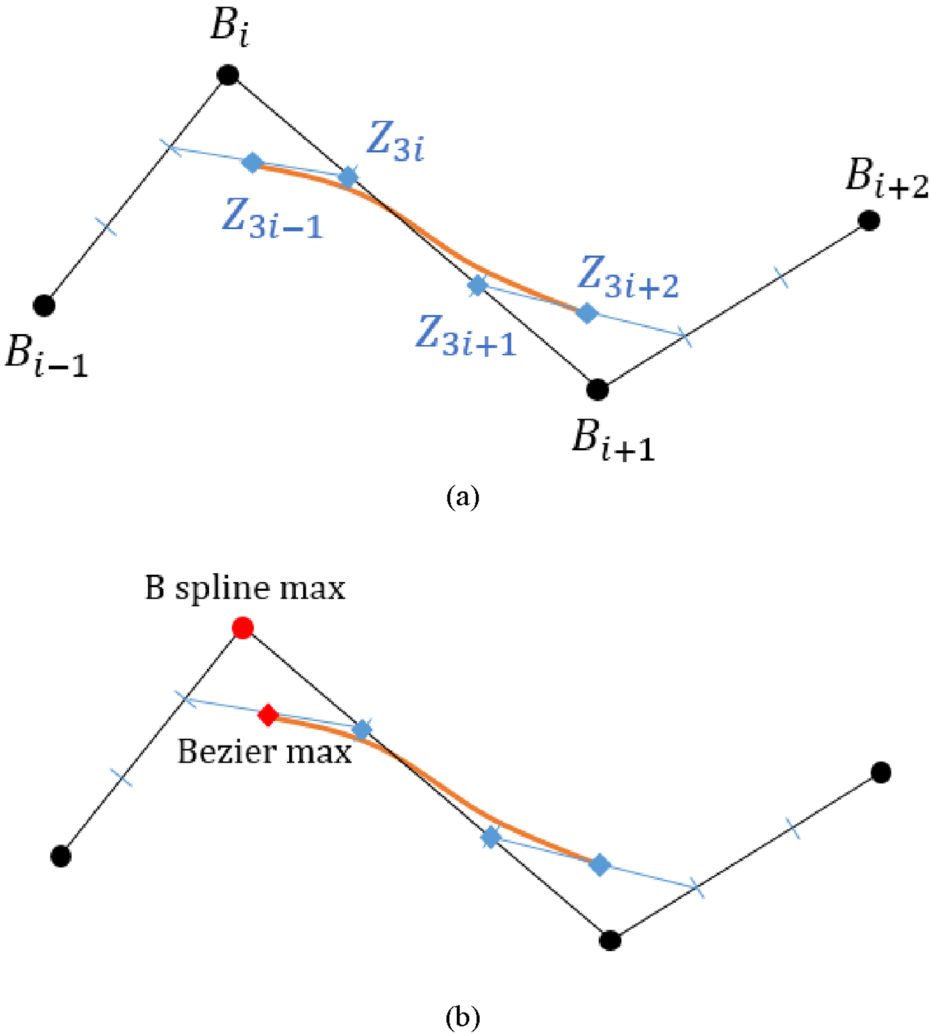


Fig. 5 **a** One spline can be drawn using both Bezier control points (Z_{3i-1}, Z_{3i}, Z_{3i+1}, and Z_{3i+2}) and B-spline control points (B_{i-1}, B_i, B_{i+1}, and B_{i+2}) **b** The maximum of the Bezier spline is smaller than the maximum of the B-spline

2.3 Empty space skipping for B-spline interpolation

After creating new volume data using B-spline interpolation, we are able to use the existing B-spline volume rendering and empty space skipping without modification. Previous methods performed an empty space skipping using a B-spline approximation [25] or B-spline interpolation [19]. Due to the convex hull property of B-splines, the maximum value of the control points was used for the maximum value of the block.

In Fig. 4(b), the height (y coordinate) values of P_{i-1}, P_i, P_{i+1}, P_{i+2}, and P_{i+3} represent the brightness values of the five consecutive voxels. Using B-spline interpolation, new control points (B_{i-1}, B_i, B_{i+1}, B_{i+2}, and B_{i+3}) and two smoothly connected

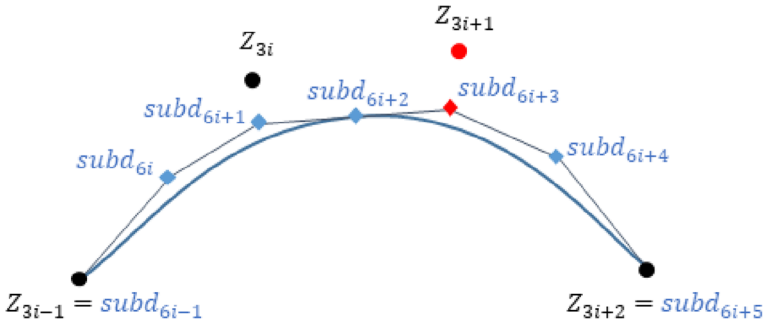


Fig. 6 Subdivision of Bezier spline in one dimension. By increasing the number of control points, we can more accurately estimate the maximum value (red points) of the curve. For our volume interpolation, this concept is extended into three dimensions

curve segments (red line) were generated. In this example, one block consists of two segments (i.e. the block size is two). Previous methods approximated the maximum of the red curve as B_{i+1} using Eq. 3.

$$\begin{aligned} segment_i.approx &= \max(B_{i-1}, B_i, B_{i+1}, B_{i+2}) \\ block.approx &= \max\left(segment_i.approx, segment_{i+1} + 1.approx \right) \\ &= \max(B_{i-1}, B_i, B_{i+1}, B_{i+2}, B_{i+3}) >= \max(\text{spline}) \end{aligned} \tag{3}$$

By using the *block.approx* instead of the *block.max* in Algorithm 1, existing methods conservatively preserve image quality, though the efficiency (possibility of skipping blocks) drops.

The maximum values of the blocks are calculated during a preprocessing step. As shown in Fig. 4, when there are two curve segments, five control points are required to generate B-splines. In the case of the volume data, the maximum value of $(s+3)^3$ control points (voxel values) is calculated for each block, where a block consists of s^3 cubic cells. Therefore, the total amount of preprocessing calculation required is O(*size of volume data*x$(s+3)^3/\ s^3$).

2.4 GPU parallelization

Parallel programming using GPUs, such as CUDA or OpenCL, includes a multi-level hierarchy of parallelism. With CUDA, the smallest parallelization unit is a thread, and 32 threads form a *warp* in which threads simultaneously execute the same instructions. The user defines the number of threads that form a thread block. Dozens of thread blocks, or thousands of threads, execute instructions in parallel to render a single image.

In volume rendering, a single thread is usually responsible for each pixel [20]. The efficiency of threads is significantly impacted under certain conditions such as branch divergence or memory coalescing [1]. To achieve the best performance, the threads in a single warp should take samples along the X-axis of the volume data [24].

The optimum shape of a tile (32 pixels), which the warp (32 threads) is responsible for, depends on the viewing direction. Zhou demonstrated that this tendency exists through various experiments [26]. Sugimoto proposed an algorithm to determine the optimum shape of the tiles

[21]. Our study proposes an improved method of determining the shape of a tile by building upon the Sugimoto's existing research.

3 Efficient space skipping using Bezier curve subdivision

3.1 Generation of Bezier control points

As described in section 2.3, empty space skipping can be performed by replacing the *block.max* in Algorithm 1 with the *block.approx* in Eq. 3. If the value of *block.approx* is

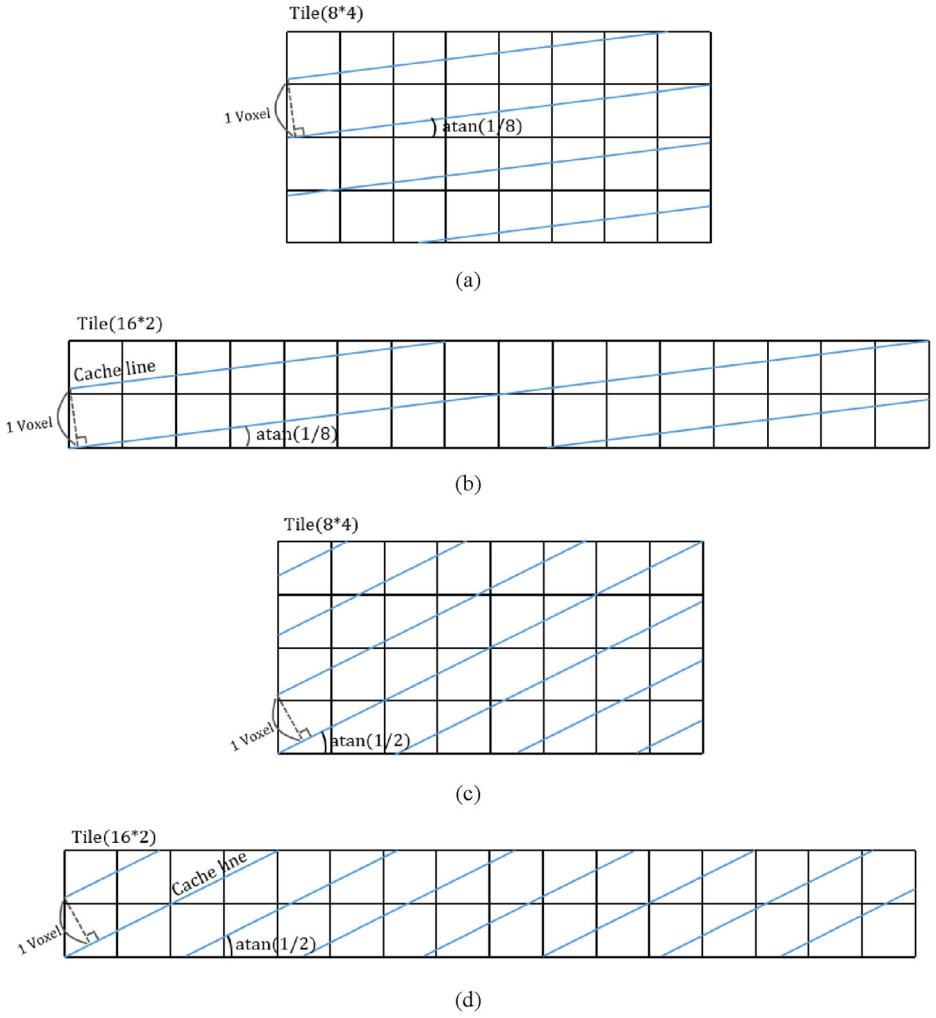


Fig. 7 Memory efficiency comparison by tile shape. If the image tile and the volume data are tilted by *arctan* (1/8), (**a**) an 8 × 4 tile requires four memory blocks (**b**) a 16 × 2 tile requires three memory blocks. If the image tile and the volume data are tilted by *arctan* (1/4), (**c**) an 8 × 4 tile requires seven memory blocks (**d**) a 16 × 2 tile requires eight memory blocks

much greater than the actual maximum value of the curve, unnecessary blocks will be designated as necessary (false positive). As a result, unnecessary blocks are not skipped and the efficiency drops [25]. Conversely, if the value of *block.approx* is smaller than the maximum of the curve, necessary blocks can be designated as unnecessary (false negative). Skipping necessary blocks creates defects in the output image. Therefore, it is important to calculate *block.approx* such that the value is larger than the curve's actual maximum but remains as small as possible.

In this study, we propose a new method that is able to skip more blocks than previous methods without the loss of image quality by transforming the B-spline into Bezier splines. We demonstrate the case of a one-variable function for simplicity. One B-spline segment can be transformed into a cubic Bezier spline as in Eq. 4.

$$
\begin{aligned}
Bspline_i(t) &= \begin{bmatrix} t^3 & t^2 & t^1 & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} B_{i-1} \\ B_i \\ B_{i+1} \\ B_{i+2} \end{bmatrix} \\
&= \begin{bmatrix} t^3 & t^2 & t^1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}^{-1} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} B_{i-1} \\ B_i \\ B_{i+1} \\ B_{i+2} \end{bmatrix} \\
&= \begin{bmatrix} t^3 & t^2 & t^1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} (B_{i-1} + 4B_i + B_{i+1})/6 \\ (2B_i + B_{i+1})/3 \\ (B_i + 2B_{i+1})/3 \\ (B_i + 4B_{i+1} + B_{i+2})/6 \end{bmatrix} \\
&= \begin{bmatrix} t^3 & t^2 & t^1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} Z_{i(-1)} \\ Z_{i0} \\ Z_{i1} \\ Z_{i2} \end{bmatrix} \\
&= Bezier_i(t)
\end{aligned}
\tag{4}
$$

The new control points for *i*-th curve segment $z_{i(-1)}$, z_{i0}, z_{i1}, and z_{i2}, are calculated from the given control points B_{i-1}, B_i, B_{i+1}, and B_{i+2} by using Eq. 5. The Bezier splines, using the new control points, are exactly the same as the previous B-spline segments.

$$
\left(z_{i(-1)}, z_{i0}, z_{i1}, z_{i2}\right) = \left(\frac{B_{i-1} + 4B_i + B_{i+1}}{6}, \frac{2B_i + B_{i+1}}{3}, \frac{B_i + 2B_{i+1}}{3}, \frac{B_i + 4B_{i+1} + B_{i+2}}{6}\right) \tag{5}
$$

The ends of the B-spline segments are connected to each other. Therefore, the last control point of the *i*-th curve of the Bezier spline overlaps with the first control point

Table 1 The most efficient rotation angle θ for each tile shape

Tile shape	The most efficient θ ($0 \leq \theta < 90°$)
32x1	*arctan* (1/32)
16×2	*arctan* (1/8)
8×4	*arctan* (1/2)
4×8	*arctan* (2)
2×16	*arctan* (8)
1×32	*arctan* (32)

Table 2 Selected tile shapes depending on the rotation angle θ

Tile shape	Rotation angle of proposed method	Rotation angle of existing method [21]
32x1	$0° \le \theta < arctan\ (1/16) = 3.576°$	$0° \le \theta < 15°$
16×2	$arctan\ (1/16) \le \theta < arctan\ (1/4) = 14.036°$	$15° \le \theta < 30°$
8×4	$arctan\ (1/4) \le \theta < arctan\ (1) = 45°$	$30° \le \theta < 45°$
4×8	$arctan\ (1.0) \le \theta < arctan\ (4.0) = 75.964°$	$45° \le \theta < 60°$
2×16	$arctan\ (4.0) \le \theta < arctan\ (16.0) = 86.424°$	$60° \le \theta < 75°$
1×32	$arctan\ (16.0) \le \theta < 90°$	$75° \le \theta < 90°$

of the $(i + 1)$-th curve. To simplify notation, we define $Z_{3i+k} \coloneqq z_{ik}$, so that $Z_{3i+2} = z_{i2} = z_{(i+1)(-1)}$ holds. One curve segment is represented using both B-spline and Bezier spline as shown in Fig. 5(a).

$$segment_i.apr\_bzr \coloneqq \max(Z_{3i-1}, Z_{3i}, Z_{3i+1}, Z_{3i+2})$$

$$\begin{aligned} block.approx >= block.apr\_bzr &\coloneqq \max\left(segment_i.apr\_bzr, segment_{i+1}.apr\_bzr \right) \\ &= \max\left(Z_{3i-1}, Z_{3i}, Z_{3i+1}, Z_{3i+2}, Z_{3(i+1)}, Z_{3(i+1)+1}, Z_{3(i+1)+2}\right) >= \max(\text{spline}) \end{aligned} \quad (6)$$

We use the *block.app_bzr* of Eq. 6 to replace the existing *block.approx*. As shown in Fig. 5(b), our *block.app_bzr* is always less than or equal to existing *block.approx* because Z_i is the weighted sum of the B_i values from Eq. 5 and the B-spline satisfies the convex hull property. As a result, we are able to skip more blocks, while avoiding the loss of image quality.

The number of Bezier control points for n curve-segments is $3n + 1$ in one dimension. The procedure can be extended to 2D or 3D volume data, where the control points generate surface-segments and cell-segments, respectively. The 1D procedure is respectively applied to the x, y, and z-axes for 3D volume data.

Because this method uses 27 (3x3x3) times as much memory as the 3D volume data, the efficient use of memory was an important consideration. In this method, only one maximum value (2 bytes) is calculated and stored for each block. We create a temporary buffer for the current blocks and then reuse buffer memory after obtaining their maximum values. This study allocated about 54 MB of buffer memory to store Bezier control points for a 2 MB area of volume data. After calculating the maximum value for the 2 MB area, we reuse the buffer memory to calculate the next 2 MB area. This computation does not affect the overall performance because it is only performed once during the preprocessing step using parallel processing in the GPU.

Table 3 Volume datasets

Name	Dimension	Pixel Size (mm)	Slice thickness (mm)	Size (MB)
ABDOMEN1 (277)	512×512×277	0.57×0.57	1.00	138.5
ABDOMEN2 (110)	512×512×110	0.63×0.63	3.00	55.0
HEAD (552)	512×512×528	0.42×0.42	1.00	276.0
LOWER (583)	512×512×583	0.78×0.78	1.50	291.5
KIDNEY1 (370)	512×512×370	0.58×0.58	1.00	185.0
KIDNEY2 (341)	512×512×341	0.60×0.60	1.00	170.5
CHEST1 (528)	512×512×528	0.66×0.66	0.75	264.0
CHEST2 (528)	512×512×528	0.66×0.66	0.75	264.0

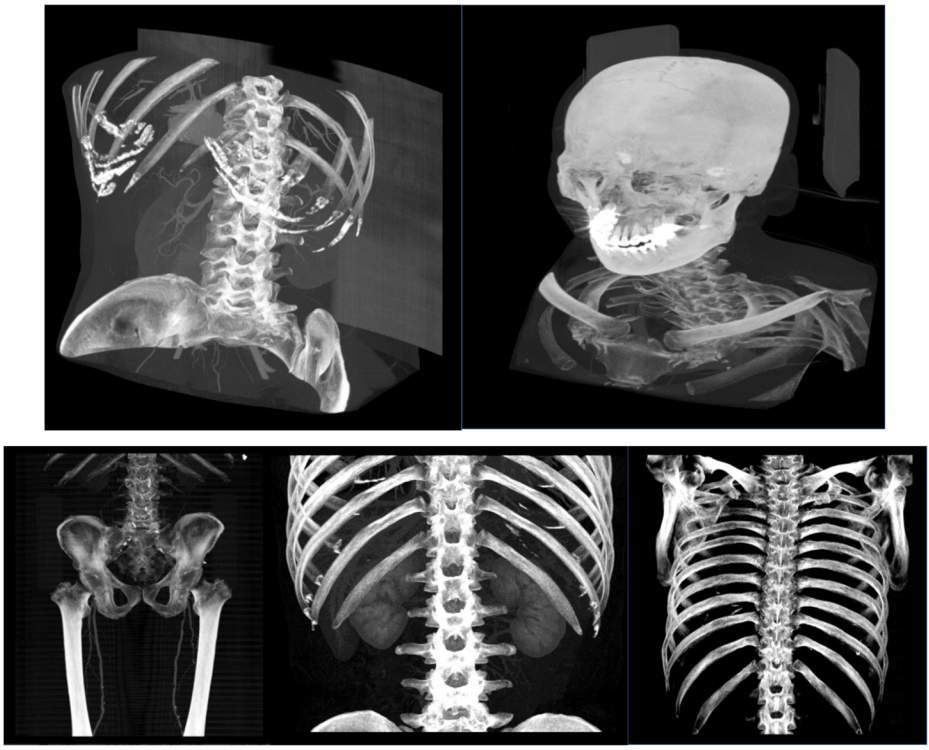


Fig. 8 Rendered images of selected volume data (ABDOMEN1, HEAD, LOWER, KIDNEY1, and CHEST1)

3.2 Subdivision of Bezier spline

A single Bezier curve segment can be divided into two or more Bezier curves by adding control points. Figure 6 shows the result after seven control points are generated from the four original control points. It is possible to skip more blocks by using the *block.app_sub* of Eq. 7, which is the maximum value of the generated control points, instead of *block.app_bzr*. For example, in Fig. 6, subd_{6i+3} is a better approximation of the maximum value of the curve than Z_{3i+1}. The Equations of the subdivision of the Bezier curve are well known and are described in Appendix 1.

$$segment_i.apr\_bzr >= segment_i.apr\_sub = \max\left(\mathrm{subd}_{6i-1}, \mathrm{subd}_{6i}, \ldots, \mathrm{subd}_{6i-5}\right) \quad (7)$$

Applying the process to 3D volume data, one cell-segment represented by 4^3 Bezier control points is subdivided into eight micro-cells with 7^3 control points. In this step, we also reused buffer memory that stores 7^3 values.

4 Efficient GPU rendering using memory coalescing

In this study, the GPU program was created using CUDA. The smallest parallel processing unit is a thread, and 32 concurrent threads constitute a warp. Each thread in a warp executes the same instruction, while they are able to access different memory addresses using their unique thread ID value. If all 32 threads in a warp

Table 4 Comparison of skip ratio for volume data and rendering time. All methods generate the same output images

Data (slices)	No acceleration		(A) Acceleration using B-spline [25]		(B) Acceleration using Bezier spline		(C) B+Subdivision		
	Rendering time (ms)	Skip ratio (%)	Rendering time (ms)	Skip ratio (%)	Rendering time (ms)	Skip ratio (%)	Rendering time (ms)	Skip ratio (%)	Rendering time enhancement (C/A)
ABDOMEN1 (277)	691.31	0	323.58	47.84	189.46	66.64	169.81	69.68	1.91x
ABDOMEN2 (110)	262.40	0	128.52	45.22	68.20	68.71	62.91	71.25	2.04x
HEAD (552)	1475.85	0	640.83	51.05	336.18	71.00	302.68	72.75	2.12x
LOWER (583)	1563.43	0	714.23	49.20	375.77	68.22	337.95	70.67	2.11x
KIDNEY1 (370)	1008.47	0	442.88	51.36	226.10	69.82	202.81	72.17	2.18x
KIDNEY2 (341)	855.83	0	381.94	50.83	211.76	68.86	189.53	71.42	2.02x
CHEST1 (528)	1423.26	0	512.94	54.74	301.37	71.26	269.64	73.73	1.90x
CHEST2 (528)	1325.94	0	525.27	53.16	305.25	70.18	275.94	72.83	1.90x

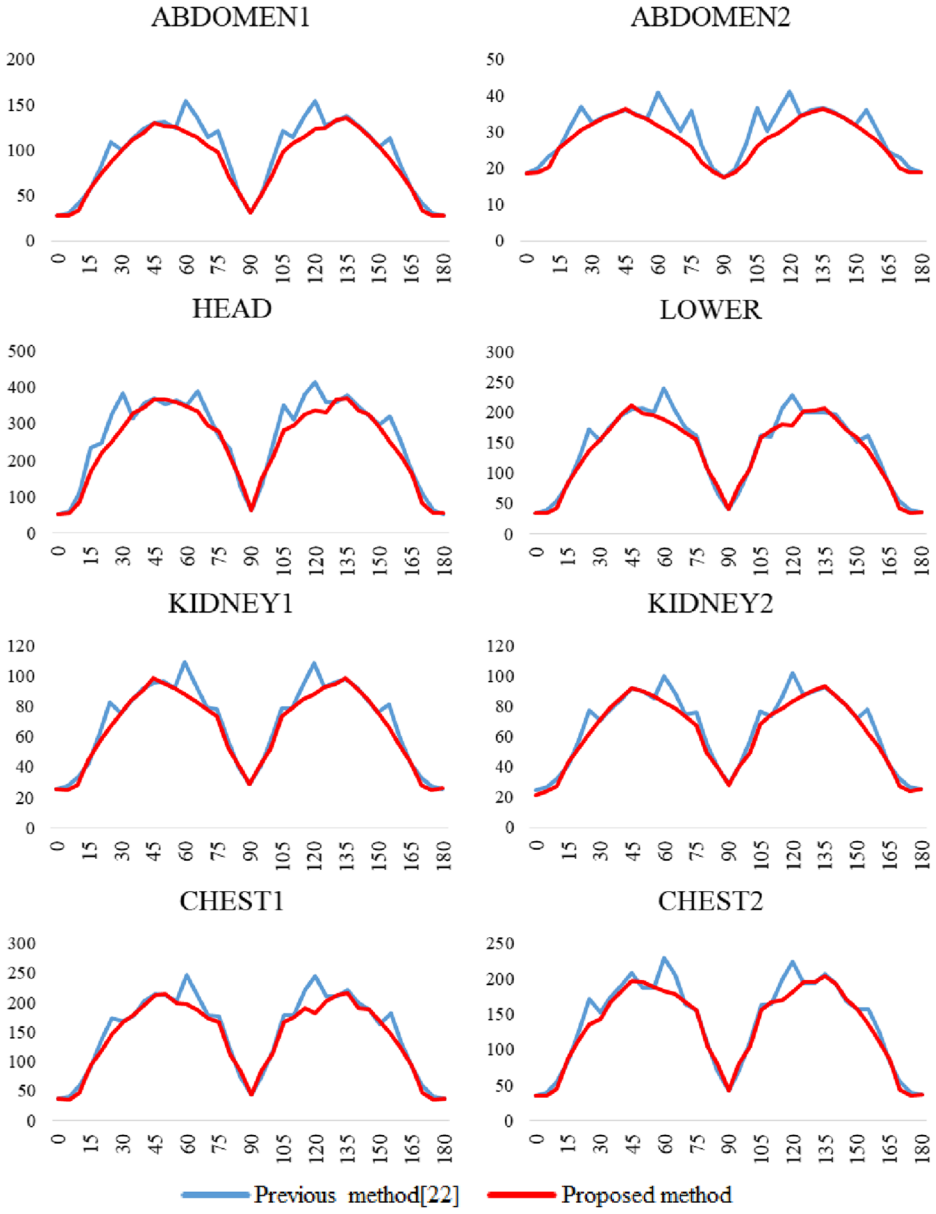


Fig. 9 Rendering time according to the viewing direction. The rendering time was measured while rotating the viewing direction about the Z-axis. In the figure, the vertical axis is the rendering time (milliseconds) and the horizontal axis is the degree of rotation (degrees)

request memory data that is close to each other, a single transfer of the memory block satisfies all threads. On the other hand, if the threads in the warp access memory that is far from each other, 32 separate memory transfers are required. To minimize this

memory divergence, we propose an advanced method that allows the threads in the warp to refer to minimal memory blocks.

Memory can be rearranged during preprocessing for better reference efficiency, however, this method is difficult to use in medical imaging systems that perform multiple functions, such as cross-sectional image extraction and segmentation [12]. Our method does not require reconfiguring the memory.

Because a single thread is responsible for each pixel, a warp processes 32 pixels simultaneously. Each rectangular region, composed of the 32 pixels that a warp is responsible for, is called a *tile*. The output image consists of tiles of the same shape because of GPU parallelism. The shape of the 32-pixel tiles must be one of the following configurations 1×32, 2×16, 4×8, 8×4, 16×2, or 32×1 (see Fig. 7). Our method determines the shape of the tiles depending on the angle between the X-axis of the output image and the X-axis of the volume data.

In Fig. 7, each tile is shown as a black rectangle composed of small squares, and the scanlines of the volume data are represented by blue lines. We can assume that the horizontal direction of each tile is parallel to the X-axis of the output image, and the GPU memory blocks (i.e. cache lines) are arranged along the X-axis of the volume data, without losing generality. Therefore, the cache lines (blue lines) are parallel to the X-axis of the volume data.

In order to supply memory to all of the threads of a tile, it is necessary to read a block of memory several times from the volume data. In Fig. 7(a), there is an 8×4 tile, where four different cache lines are transferred when a warp refers to the memory of the volume data. In the same situation, using a 16×2 tile is more efficient because it only requires three cache lines as shown in Fig. 7(b). Even though the spacing of the blue lines is dependent on the image magnification selected by the user, the relative comparison is the same.

Because the viewing direction changes according to the user's input, the angle between the black tile and the blue cache lines also changes. Because the 8×4 tile in Fig. 7(c) requires seven memory references while the 16×2 tile of Fig. 7(d) requires eight, the optimal tile shape should be calculated for every frame. We regard determining the best tile shape as filling the tile with the minimum number of tilted parallel lines.

When the angle between the X-axis of the image and the X-axis of the volume data is θ, the shape of each tile and corresponding rotation angle for maximum efficiency are presented in the Table 1. It is ideal when the *width* / *height* value of the tile is equal to $tan\ \theta$, as shown in Fig. 7(b) and (c). For reference, a mathematical explanation is given in Appendix 2.

Next, the most efficient shape of tiles, corresponding to the given θ from the user input, should be determined. Table 2 presents a comparison between our method and the previous method [21]. The previous study determined the shape of the tiles according to the rotation angle θ without basis in mathematics, while our method is based on the proof given in Appendix 2. As a result, when $\theta = 25°$, for example, the existing method selects a 16×2 tile, while the proposed method selects an 8×4 tile.

When developing a software system, the shape of the tiles is determined by selecting the shape of a CUDA thread block. For example, if the efficient tile is 4×8, we determine the shape of the thread block to be 4×64 (assuming the thread block is composed of 256 threads) [21].

5 Experimental result

The experiments were conducted on a PC equipped with an Intel Core i5 CPU, 8 GB RAM, and a GeForce GTX 960 GPU. We implemented our method using Visual studio C++ and CUDA on a Windows 10 operating system. The volume datasets are presented in Table 3, and the rendered images from selected datasets are shown in Fig. 8. We implemented B-spline interpolation MIP volume rendering with empty space skipping without quality loss. Because the characteristics of the B-spline are fully elucidated in existing studies [11, 17, 23, 25], no relative comparison of image quality was performed.

Table 4 shows the improvement in speed of the proposed method as compared to other methods. The visualization time of the existing method and the proposed method were measured for each dataset. The rendering time is the average value when the viewing direction is rotated about the Z-axis.

By using the existing B-spline skipping method [25] (refer section 2.3), we were able to skip approximately 45 ~ 51% of the data (column A). By using the proposed Bezier spline method, 66 ~ 71% of data can be skipped with no loss in image quality (column B). The subdivision method discussed in section 3.2 was used to more accurately calculate the maximum density values. The skip ratio was increased to 69 ~ 73% using subdivision (column C). The rendering speed of the subdivision method was increased by about 10% despite the slight improvement (2 ~ 3%) of the skip ratio. As a result, the proposed method outputs the same image as existing methods [25] and improves the rendering speed by a factor of 2. Practical volume data can be visualized at an interactive speed.

Rendering time is directly related to the size of the volume data, and the skip ratio is related to the characteristics of the distribution of volume data. For example, if a skeletal section with a large voxel value occludes an air section with a small voxel value, the skip probability is increased.

To demonstrate the performance of the tile shape selection method proposed in section 4, we compared the rendering time of the proposed method with the previous method [21]. The experiment was based on our subdivision method (Table 4 column C). Additionally, we added a simple optimization method.

For each pixel, the center of each ray is sampled when the ray starts, taking into consideration that the value in the volume center of medical images is relatively large. This does not affect the MIP image quality [22] because the maximum value among samples along the ray is calculated. We are able to skip more blocks because the ray starts with a large cumulative value in Fig. 3 [5, 6]. Although the existing method [21] is separate from the acceleration proposed above, all accelerations were applied equally to the proposed and existing methods to measure the effectiveness of the method proposed in section 4.

Figure 9 shows the rendering times, which were measured by rotating the viewing direction around the Z-axis. As seen from this figure, the proposed method was more efficient than the existing method. Although the performance improvement of proposed method is not large, this study is significant in that it demonstrates the method's ability to calculate the best tile shape theoretically and then verified it experimentally.

Rendering time is able the best when the axis of the volume data is parallel to the axis of the image, and performance is degraded when the axes are oblique. This occurs because memory references are not efficient, which is seen in most object-order volume visualizations [7].

6 Conclusion

This paper proposed a high-quality MIP rendering technique for medical visualization systems. In order to perform cubic interpolation at high speed, we efficiently skip the empty space in volume data. We proposed a Bezier spline-based calculation method that adds control points in order to calculate the maximum value of each block. The convex hull property of the Bezier spline allows this acceleration without the loss of image quality. In addition, spline subdivisions can be used to more accurately calculate the maximum values and skip more blocks. As a result, we have achieved 2 times the performance without changing the image quality, as compared to the existing method. In addition, the additional memory usage was limited by reusing block memory.

We also proposed a method to improve the efficiency of memory reference when performing visualizations with the GPU. The thread group of GPUs executed in parallel sees improved performance when all referring to nearby memory. The most efficient tile shape was determined depending on the viewing direction. The proposed method was described theoretically and the experimental results confirmed that it is an improvement over the existing method.

This study was limited in that section 3 does not consider the numerical error of floating-point operations. It is worth noting that subdivision takes a long time, although it is performed only once. In section 4, the rendering speed variation can be large depending on the direction of observation. In the future, this research can be extended to general direct volume rendering using transfer functions.

Appendix 1

Subdivision of Bezier curve

$$
\begin{aligned}
subd_{6i-1} &= Z_{3i-1} \\
subd_{6i} &= \frac{Z_{3i-1} + Z_{3i}}{2} \\
subd_{6i+1} &= \frac{Z_{3i-1} + 2Z_{3i} + Z_{3i+1}}{4} \\
subd_{6i+2} &= \frac{Z_{3i-1} + 3Z_{3i} + 3Z_{3i+1} + Z_{3i+2}}{8} \\
subd_{6i+3} &= \frac{Z_{3i} + 2Z_{3i+1} + Z_{3i+2}}{4} \\
subd_{6i+4} &= \frac{Z_{3i+1} + Z_{3i+2}}{2} \\
subd_{6i+5} &= Z_{3i+2}
\end{aligned}
$$

Appendix 2

Proof of Tables 1 and 2

We prove the Table 1. “The most efficient tile shape is w x h if rotation is arctan (h/w).”

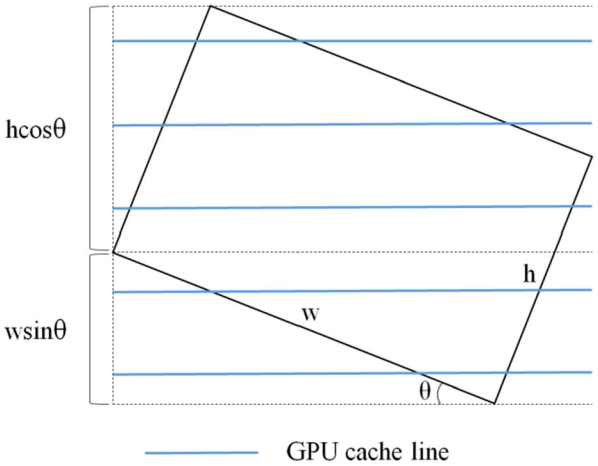
Memory transfer time is proportional to the height of rotated tile (HRT). We define the HRT when the tile width is w and the tile height is h as $HRT_{w,\ h} \coloneqq h\cos\theta + w$

$\sin\theta$ (*assuming* $0 \le \theta < \pi/2$). The HRT is minimized when arctan (h/w) = θ, because:

$$\min_{\theta} HRT_{w,h} = \min_{\theta} h\cos\theta + w\sin\theta$$

$$minimum\ at: h\cos\theta = w\sin\theta$$

$$\tan\theta = \mathrm{h}/\mathrm{w}$$



Now we prove the first row of Table 2. "32x1 tile is efficient when $0 \le \theta <$ arctan(1/16)".

If θ is close to 0, we obviously select 32x1 tile. As θ increases, we select the 32x1 tile if $HRT_{32,1} < HRT_{16,2}$ and select the 16×2 tile if $HRT_{32,1} > HRT_{16,2}$. The boundary θ value between them is when $HRT_{32,1} = HRT_{16,2}$. We select the 32x1 tile when $0 \le \theta <$ arctan(1/16) because:

$$1\cos\theta + 32\sin\theta = 2\cos\theta + 16\sin\theta$$

$$\theta = \arctan(1/16)$$

Acknowledgments This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(Ministry of Science and ICT) (No. 2017R1E1A1A03070494).

Code availability Not applicable.

Funding This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(Ministry of Science and ICT) (No. 2017R1E1A1A03070494).

Data availability Not applicable.

Declarations

Conflicts of interest/competing interests Not applicable.

References

1. Champagnat F, Le Sant Y (2012) Efficient cubic b-spline image interpolation on a GPU. J Graphics Tool 16(4):218–232
2. de Boor C (1978) A Practical guide to splines. Springer-Verlag, New York
3. Eklund A, Dufort P, Forsberg D, La Conte SM (2013) Medical image processing on the GPU – past, present and future. Med Image Anal 17(8):1073–1094
4. Kaufman AE, Mueller K (2005) Overview of volume rendering. The Visualization handbook 7:127–174
5. Kwon O, Kang ST, Kim SH, Kim YH, Shin YG (2015) Maximum intensity projection using bidirectional compositing with block skipping. J X-ray Sci Technol 23(1):33–44
6. Kye H, Sohn BS, Lee J (2012) Interactive GPU-based maximum intensity projection of large medical data sets using visibility culling based on the initial occluder and the visible block classification. Comput Med Imaging Graph 36(5):366–374
7. Lacroute P, Levoy M (1994) Fast volume rendering using a shear-warp factorization of the viewing transformation. In: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques. pp. 451–458
8. Levoy M (1988) Display of surfaces from volume data. IEEE Comput Graph Appl 8(3):29–37
9. Levoy M (1990) Efficient ray tracing of volume data. ACM Trans Graph 9(3):245–261
10. Marschner SR, Lobb RJ (1994). An evaluation of reconstruction filters for volume rendering. In: Proceedings Visualization'94, IEEE, pp. 100–107
11. Mihajlovic Z, Budin L, Radej J (2004) Gradient of B-splines in volume rendering. IEEE Mediterranean Electrotechnical Conference. IEEE, In, pp 239–242
12. Misaki Y, Ino F, Hagihara K (2017) Cache-aware, in-place rotation method for texture-based volume rendering. IEICE Trans Inf Syst 100(3):452–461
13. Mora B, Ebert D.S (2005) Low-complexity maximum intensity projection. ACM Trans Graph 24(4):1392–1416
14. Mroz L, König A, Gröller E (1999) Real-time maximum intensity projection. In: Data Visualization'99, springer, pp 135-144
15. Mroz L, Hauser H, Gröller E (2000) Interactive high-quality maximum intensity projection. Comput Graphics forum 19(3):341–350
16. Nehab D, Maximo A, Lima S, Hoppe H (2011) GPU-efficient recursive filtering and summed-area tables. ACM Trans Graph 30(6):176
17. Ruijters D, Thévenaz P (2010) GPU prefilter for accurate cubic B-spline interpolation. Comput J 55(1):15–20
18. Schreiner S, Galloway RL Jr (1993) A fast maximum-intensity projection algorithm for generating magnetic resonance angiograms. IEEE Trans Med Imaging 12(1):50–57
19. Shin Y, Kye H (2016) High quality volume visualization using B-spline interpolation. J Korea Comput Graphics Soc 22(3):1–9 (in Korean)
20. Smelyanskiy M, Holmes D, Chhugani J, Larson A, Carmean DM, Hanson D, Dubey P, Augustine K, Kim D, Kyker A, Lee VW, Nguyen AD, Seiler L, Robb R (2009) Mapping high-fidelity volume rendering for medical imaging to CPU, GPU, and many-core architectures. IEEE Trans Vis Comput Graph 15(6):1563–1570
21. Sugimoto Y, Ino F, Hagihara K (2014) Improving cache locality for GPU-based volume rendering. Parallel Comput 40(5):59–69
22. Sun Y, Parker DL (1999) Performance analysis of maximum intensity projection algorithm for display of MRA images. IEEE Trans Med Imaging 18(12):1154–1169
23. Unser M, Aldroubi A, Eden M (1993) B-spline signal processing: part II - efficient design and applications. IEEE Trans Signal Process 41(2):834–847
24. Wang J, Yang F, Cao Y (2017) A cache-friendly sampling strategy for texture-based volume rendering on GPU. Visual Inform 1(2):92–105
25. Zhang C, Xi P, Zhang C (2011) CUDA-based volume ray-casting using cubic B-spline. In: 2011 International Conference on Virtual Reality and Visualization, IEEE, pp. 84-88
26. Zhou D, Du H, Zhao F, Kan H, Li G, Qiu B (2016) Improving efficiency for CUDA-based volume rendering by combining segmentation and modified sampling strategies. Int J Simul Syst, Sci Technol 17(42):1–9