J Supercomput (2014) 67:324–344
DOI 10.1007/s11227-013-1004-x

Direct volume rendering of unstructured tetrahedral meshes using CUDA and OpenMP

Erhan Okuyan · Uğur Güdükbay

Published online: 31 August 2013

Abstract Direct volume visualization is an important method in many areas, including computational fluid dynamics and medicine. Achieving interactive rates for direct volume rendering of large unstructured volumetric grids is a challenging problem, but parallelizing direct volume rendering algorithms can help achieve this goal. Using Compute Unified Device Architecture (CUDA), we propose a GPU-based volume rendering algorithm that itself is based on a cell projection-based ray-casting algorithm designed for CPU implementations. We also propose a multicore parallelized version of the cell-projection algorithm using OpenMP. In both algorithms, we favor image quality over rendering speed. Our algorithm has a low memory footprint, allowing us to render large datasets. Our algorithm supports progressive rendering. We compared the GPU implementation with the serial and multicore implementations. We observed significant speed-ups that, together with progressive rendering, enables reaching interactive rates for large datasets.

Keywords Direct volume visualization · Volume rendering · Unstructured tetrahedral meshes · Graphics Processing Unit (GPU) · Compute Unified Device Architecture (CUDA) · OpenMP

1 Introduction

Volume visualization is useful in many areas. Medical fields extensively benefit from this method, and computational fluid dynamics uses it to inspect several properties of fluid flow. In general, any discipline that studies the internal structure of a volume

E. Okuyan · U. Güdükbay (✉)
Department of Computer Engineering, Bilkent University, Bilkent, 06800 Ankara, Turkey
e-mail: gudukbay@cs.bilkent.edu.tr

E. Okuyan
e-mail: okuyan@cs.bilkent.edu.tr

benefits from volume visualization. Volumetric data can be represented in different forms, depending on the application and data-capture technologies. We focus on directly rendering unstructured tetrahedral meshes where volume's interior needs to be visualized. There are many approaches to rendering unstructured grids and accuracy is usually important. Ray-casting-based methods, which our work focuses on, are widely accepted. These techniques provide accurate results but are computationally costly; many actual volume datasets contain millions of tetrahedra. Rendering such complex data in a timely manner is a real challenge.

We propose direct volume rendering (DVR) algorithms for parallel architectures that achieve interactive rates for large datasets with good image quality and memory efficiency. We modified the *cell-projection* algorithm described in [1] to exploit parallelization and used OpenMP to parallelize the implementation for multi-core systems. We extended our *cell-projection* algorithm for GPUs using CUDA and focused on enhancing the highly parallelizable characteristic of the algorithm.

1.1 Motivation

There are many works done on volume rendering. A large proportion of recent volume renderers are based on shader programming in order to achieve high rendering rates. Although such approaches are fast, they have various drawbacks. First of all, shader programming is quite restrictive. There are memory and instruction limitations. The total memory available to each shader unit is quite low. Also, the available instruction set is limited and the total number of instructions in a shader program has an easily reachable upper bound. Accordingly, many algorithms are too complex to be implemented with a shader program. Thus, shader programming based volume renderers usually use approximation in order to satisfy shader restrictions, leading to inferior graphical quality. Also, those volume renderers have very limited support to incorporate additional features, such as LOD approaches, lighting effects, and isosurface effects.

On the other hand, CPU based volume renderers have great flexibility. Such restrictions on shader programming based renderers do not exist for these renderers. Very accurate images can be rendered and there are no restriction on adding new features. However, CPU based volume renderers are much slower than shader programming based renderers. They cannot give enough performance to be used interactively.

Our main motivation was to develop a volume renderer without the restrictions of shader based renderers and is much faster than CPU based renderers. Our goal was to focus on image quality first. We performed the computations as accurately as possible. For example, some shader programming based volume renderers use preintegration tables, represented as 3D textures, to compute the effects of tetrahedra. The size limitation on the 3D texture limits the accuracy of the computation. On the other hand, we perform the actual computation leading to accurate results. We also did not allow any visual artifacts (to achieve high rendering rates, some renderers allow visual artifacts). Our second goal is to reach interactive rates while rendering decent image resolutions. We developed a CUDA based volume renderer to satisfy both of our goals. CUDA provides a rich programming environment that allows the implementation of complex algorithms. Thus, we can implement accurate rendering algorithms and produce high quality images. Since CUDA provides GPU acceleration, the performance would be much higher than CPU implementations. In order to improve the interactive usability, we also supported progressive rendering. Progressive rendering allows rendering the volume in low-resolution first and then progressively improves the image to the desired resolution. Since the low-resolution image can be rendered much faster, it can be displayed much earlier, improving the interactivity significantly. Progressive rendering is particularly useful for very high resolutions, where rendering takes much longer.

Our volume renderer can be integrated with additional features, like lighting and iso-surface effects. The modular architecture and the flexible programming environment provided by CUDA allow the implementation of complex algorithms and easy integration. Our volume renderer can also be integrated with level-of-detail (LOD) approaches. It can be used in a dynamic view-dependent refinement setup [11], where the volumetric data can be selectively refined based on viewing parameters. View-dependent refinement can significantly reduce rendering costs without noticeable degradation of image quality.

Memory efficiency is crucial for volume renderers, especially if they are GPU accelerated. Our implementation focuses on keeping the memory overhead as low as possible without significantly affecting the performance. We also employed mechanisms to allow rendering a volume in several iterations, reducing the memory requirement significantly. Accordingly, our volume renderer can handle large volume datasets.

2 Related work

Volume rendering is well studied. There are two main types of volume data: regular and irregular grids. Regular grid representations are widely used in medical imaging, with texture-based techniques dominating. Earlier approaches sampled volume with parallel planes along the view direction [9, 12]. The nature of graphics card allows storing the volume data in the GPU as 3D textures; Ertl et al. used a preintegration approach to efficiently render volume using 3D texture representation [4, 13].

Although regular grids can be efficiently rendered, they can be large, limiting the detail level of the volume data. Unstructured grids can be represented in a much more compact form, thus they can reach much higher detail levels. Iso-surface techniques allow fast rendering of volume data, which can be useful if surfaces are the critical regions in the volume. Lorensen and Kline proposed the Marching Cubes algorithm [10], which became the basis for many later algorithms.

In our work, we focus on direct volume rendering algorithms, of which visibility ordering is an important part. If the mesh primitives (faces or polyhedra) are ordered in a way that no primitive is occluded by an earlier primitive in the list, the list is visibility ordered. Such lists can be efficiently rendered by graphics cards. Cook et al. [8] and Kraus and Ertl [3] proposed methods for efficient visibility sorting. Shirley and Tuchman proposed a projected tetrahedra algorithm [15], which was later extended to GPUs using vertex shaders by Wylie et al. [18].

Garrity [5] and Koyomada [7] exploited connectivity to achieve fast cell traversals. This approach was later extended to GPUs by Weiler et al. [17], where the mesh and the connectivity information are represented as 3D and 2D textures, respectively. Callahan et al. introduced a visibility ordering algorithm, HAVS [2], which performs a rough sorting on the CPU and finalizes the sorting in the GPU. The initial CPU sorting phase sorts the face primitives according to their center-to-eye distances. The resulting list contains errors, but they are corrected in the GPU using the *k-buffer* approach. See Silva et al. [16] for an extensive survey of volume rendering techniques.

3 Cell-projection algorithm

Direct volume rendering is a computationally-intensive process. Early algorithms focused on single-processor environments, then the trend shifted to parallel algorithms for PC clusters. With the recent developments in multicore CPUs and GPU-based computing techniques, volume rendering algorithms for these platforms have increased in importance because single CPUs are not powerful enough to render even simple volume data into reasonable resolutions in acceptable time frames. We based our work on a well-known direct volume rendering algorithm: the cell-projection algorithm.

The cell-projection algorithm is simple yet efficient; it does not rely on tetrahedra's neighborhood information to order them. The data representation requirement is low and the data access patterns are relatively uniform, making this algorithm suitable for GPU implementation. In this section, we present the single-core version of the cell-projection algorithm. We describe the data types used by the algorithm and then outline its steps. Then we describe the implementation of the algorithm on multicore systems using OpenMP.

3.1 Data structures

Volume data used in real-world application have grown, with datasets of tens of millions of tetrahedra quite common. This growth has also increased memory requirements for DVR implementations; to satisfy such requirements with common computer configurations, the data structures used in this work are kept as minimal as possible:

- *Vertex*: Vertex structure contains the coordinate and scalar value associated with the vertex. Float values are used to store these data, making the vertex structure 16 bytes in size.
- *Tetrahedron*: Tetrahedron structure contains indices of four vertices, which comprise the tetrahedron. Integer values are used to store index values, making the tetrahedron 16 bytes in size. The size can be reduced with encoding. Each index value is a decimal value between zero and the number of vertices in the dataset. Thus $\lceil \log_2(\mathit{NumberOfVertices}) \rceil$ bits are enough to represent an index value. Usually, 24 bits are sufficient to represent a vertex index of a sizable volume dataset, making it possible to reduce the tetrahedron size 25 % or more, depending on the number of vertices. Processing an unencoded index value is much faster, however, because index values are extensively used. For that reason, we decided to use the integer type to store index values to avoid such encoding overheads.

- *Intersection record*: Direct volume rendering algorithms throw a ray from each pixel. As the ray travels through the volume, it intersects with several tetrahedra. The effects of such intersections are used to determine the pixels' final color. The intersection record consists of the pixel value from where the ray has been thrown and the tetrahedron's index that the ray intersected with. Because all the intersection records have to be created before they are consumed, their total size can be very large, and encoding the intersection record is necessary. An intersection record can be represented with $\lceil \log_2(\mathit{NumberOfTetrahedra}) \rceil + \lceil \log_2(\mathit{NumberOfPixels}) \rceil$ bits. To store individual records, this size should be rounded up to a multiple of eight bits. A record size of six bytes is sufficient for all the datasets and resolutions tested in this implementation.
- *Per-pixel intersection lists*: The procedure that extracts the intersection records groups them according to their pixel values, which is done via per-pixel intersection lists. This structure is simply a collection of intersection record arrays, one per each pixel value. As these arrays can increase their sizes dynamically, this structure does not introduce too much memory overhead.
- *Intersection effect*: An intersection effect structure is produced after an intersection record is processed. It includes the eye distance to the first point that the thrown ray hits on the tetrahedron. This value is used to sort the intersection effects. The color effect left by the tetrahedron on the intersecting ray is also recorded. As the color is stored as four floats, representing the *rgba* values, the size of this structure is 20 bytes. Because the structure is created and destroyed on a per-pixel basis, its effect on the memory requirement is low.
- *Color map*: A color map is the tabulated form of the transfer function. It is an array of color values. A minimum scalar value is assigned to the first entry and a maximum scalar value is assigned to the last entry. The scalar values associated with the remaining entries are calculated by interpolating the minimum and maximum scalars. Color entries are found by using the transfer function with the associated scalars.

3.2 Algorithm

The overall flow of the single-core version of the cell-projection algorithm is given in Algorithm 1. First, screen space coordinates of the vertices are calculated and the intersection records are extracted. Then calculating, sorting, and compositing intersection effects occur in a sequence for every pixel. The steps of the algorithm are described below.

Screen space projection of vertices In this step, vertices are projected onto the screen according to the view parameters, and their screen space coordinates are calculated. These coordinates are stored in the *SSC* array, which is then used during tetrahedron projections.

Extracting intersection records This step can be considered the screen space projection of the tetrahedra. The algorithm calculates screen space projections of each tetrahedron. Rays thrown for any pixel under the projection of a tetrahedron intersect

```
CellProjectionAlgorithm()
begin
    IntersectionRecord *$PerPixelIntersectionLists[Width][Height]$;
    $SSC$ = ComputeScreenSpaceProjections($Vertices$);
    ExtractIntersectionRecords($Tetrahedra$, $Vertices$, $SSC$, $PerPixelIntersectionLists$);
    for $i = 0$ upto $Width$ do
        for $j = 0$ upto $Height$ do
            $list = PerPixelIntersectionLists[i][j]$;
            $IntEffctList$ = CalculateIntersectionEffects($list$);
            SortIntersectionEffects($IntEffctList$);
            Color $c$ = ReduceIntersectionEffects($IntEffctList$);
            Image$[i][j] = c$;
end
```

Algorithm 1: Cell-projection algorithm

with the tetrahedron. Thus, a tetrahedron will contribute to the colors of the pixels under its projection. The cell-projection algorithm extracts any such tetrahedron-pixel pairs (the intersection records). Intersection records are stored in a pixel index-based array, called *PerPixelIntersectionLists*.

Intersection records are found by traversing each tetrahedron; the algorithm calculates its screen space projections by projecting its four vertices. When the projection points of these vertices are connected, a triangle or a quadrilateral will be formed, and with a basic scan-line algorithm, the pixels covered by this projection area can be calculated. The tetrahedron id and the pixel values are then encoded into an *IntersectionRecord* and inserted into the *PerPixelIntersectionLists*.

Calculating intersection effects When a thrown ray travels through a tetrahedron, it loses intensity and its color is affected. The effects of all the tetrahedra that a ray has traveled through can be combined to obtain the final effect on the ray. This procedure (Algorithm 2) uses the intersection record list for the current pixel and calculates the intersection effects for each record in the list individually.

Algorithm 2 calculates the intensity and color of a tetrahedron intersection on the ray. The first step calculates two intersection points (ip_0 and ip_1) of the ray and the input tetrahedron. Let ip_0 be the closer point to the eye. The distance between the eye point and ip_0 is recorded in the intersection effect record; this value will be used in the sorting phase. The ray's path, which is the line segment between ip_0 and ip_1, is divided into a predefined number (*NumOfSamples*) of line segments. At the starting point of these line segments, the scalar value is calculated with interpolations using the vertices of the tetrahedron. The color of this point is determined by the *ColorMap* table. As the ray travels from ip_0 to ip_1, the color and intensity contributions can be approximated. We describe the interpolation process by the following example:

Let ip_0 be (10, 10) and ip_1 be (22, 19). Dividing the path into three segments, the points we are interested in are ip_{01} = (14, 13) and ip_{02} = (18, 16). The algorithm approximates the path that the ray travels with three line segments of length five: [ip_0,

```
CalculateIntersectionEffects($Tetrahedron\ t$, $Pixel\ p$)
begin
    IntersectionEffect $record$;
    Ray $*R$ = new Ray($p$);
    [$ip_0$, $ip_1$] = RayTetrahedronIntersection($R$,$t$);
    $record$.dist = $|ip_0 - EYE|$;
    $d = \frac{|ip_1 - ip_0|}{NumOfSamples}$;
    Color $c = [0, 0, 0, 0]$;
    $record$.color = $[0, 0, 0, 0]$;
    for $i = 0$ to $NumOfSamples$ do
        Point $ip = ip_0 + d \times i$ ;
        $s$ = InterpolateScalar($t$,$ip$);
        $c$ = getColorFromScalar($s$);
        for $j = 0$ to 4 do
            $record$.color[$j$] $+= DistConst \times d \times (1 - record$.color[3]$) \times$
            $record$.color[$j$];
    return $record$;
end
```

Algorithm 2: Calculating the effect of the intersection between a ray and a tetrahedron on the ray

ip_{01}], [ip_{01}, ip_{02}], and [ip_{02}, ip_1]. The ray starts traveling with full intensity and each segment is assumed to have a uniform color, which is the color at the beginning of the segment. The color contributions of the line segments to the pixel are proportional to the color attributes of the traveled region, the travel distance, the opacity coefficient of the region and the intensity of the ray itself. The ray loses most of its energy while traveling through nontransparent regions; thus later regions have a lesser effect on the final color. After the ray travels through all regions, its final color is recorded.

Accurately calculating the scalar value of a point on a tetrahedron is important because poor interpolations may cause significant artifacts. The interpolation process is given in Algorithm 3. It starts by selecting a reference vertex, which can be any one of a tetrahedron's vertices. Then the M matrix, which contains the positions of the other three vertices of the tetrahedron relative to the reference vertex, is calculated. The N vector stores the relative position of the input point to the reference vertex. The scalar vector (*Scalar*) contains scalar value differences of the vertices relative to the scalar values of the reference vertices. Then the equation $M \times R = Scalar$ is solved to obtain the R vector, which represents a coefficient vector that will give the relative scalar value of a point when multiplied with the relative position vector of that point. The `SolveEquation` function calculates this coefficient, dot product of the relative position vector of point (N) and the coefficient vector (R). By adding the scalar value of the reference vertex, the interpolated scalar is found.

Sorting intersection effects This step is achieved using the *dist* parameter. The number of elements to sort is usually in the low hundreds. Although many sorting algo-

```
InterpolateScalar(Tetrahedron $t$, point $p$);
begin
    float $M[3][3]$;
    float $Scalar[3]$;
    float $N[3], R[3]$;
    float $s$;
    for $i = 0$ upto 3 do
        $M[i] = t$.V$[i + 1]$.coords $- t$.V[0].coords;
    $N = p - t$.V[0].coords;
    for $i = 0$ upto 3 do
        $Scalar[i] = t$.V$[i + 1]$.scalar $- t$.V[0].scalar;
    SolveEquation($M$, $Scalar$, $R$);
    /* Solves the equation $M \times R = Scalar$ for $R$. */
    $s = R \cdot N + t$.V[0].scalar;
    return $s$;
end
```

Algorithm 3: Interpolation of scalar value within a tetrahedron

```
ComposeIntersectionEffects(IntersectionEffect $*list$);
begin
    Color $c = [0, 0, 0, 0]$;
    for $i = 0$ upto $list.length$ do
        Color $r = list[i]$.color;
        for $j = 0$ to 3 do
            $c$.color$[j]$ += $(1 - c$.color[3]$) \times r$.color$[j]$;
        if $c$.color[3] >= 0.9999 then
            break;
    return $c$;
end
```

Algorithm 4: Composition of intersection effects

rithms can be used for this job, the size of the list favors some of them; we found that quicksort performs well for the size range of the intersection lists.

Compositing intersection effects Sorting the intersection effects by the distance of the first intersection point to the eye orders the tetrahedra by the ray's visit order. Algorithm 4 describes the composition of the contributions along the ray. If the opacity of accumulated color exceeds a predefined threshold, the ray terminates because the remaining tetrahedra will have no significant effects (early ray termination).

3.3 Multi-core implementation with OpenMP

The cell-projection algorithm is highly suitable for parallelization, because for the most part, memory accesses are structured, and race conditions are thus avoided. OpenMP provides a useful interface for parallelization on multicore processors with a shared memory architecture. It divides the workload among threads and then executes them through different cores. Since the cores use the shared memory, there are no data transfer issues as there are with parallel clusters.

OpenMP also supports parallelization of for-loops by distributing the iterations among threads. This approach works unless one iteration requires data produced by another iteration or there are race conditions among iterations. Screen space projection of vertices can be parallelized trivially, since no race conditions exist. The for-loops that process intersection records for each pixel can also be parallelized, but as the iterations in these loops use some temporary data that data should be replicated for each thread to avoid conflicts.

Extracting intersection records necessitates traveling through the tetrahedra and inserting intersection records to the per-pixel intersection lists under a tetrahedron's projection. Since different tetrahedra can have projections on the same pixel, a race condition on that pixel's intersection list is possible. To avoid this scenario, the *Per-Pixel Intersection Lists* structure should be replicated. After the extractions are complete, the lists of each thread can be combined. While this approach is amenable to OpenMP parallelization, we observed that it does not improve computation time significantly; thus, we used the serial version.

4 Cell-projection algorithm on GPU

Although the CPU-based cell-projection algorithm's performance is reasonable for small datasets, it is not sufficient for large datasets. However, this algorithm is well suited to a GPU implementation, as it focuses on one group of data at a time and its memory accesses are structured. Further, as the algorithm is well structured in terms of execution flow, it can be efficiently parallelized.

We analyzed some of the volume rendering tools that use the k-buffer approach. With this approach, the visibility sorting of the faces starts by presorting them on the CPU according to the distance between the face center and the eye or some other similar criteria. Then the visibility sorting can be completed by the shaders. The approach relies on the presorting being accurate enough so that a k-sized buffer stored on shaders can correct the visibility order. We analyze various volume datasets and observe cases where k values higher than 30 are required to obtain correct visibility ordering. The speed of the k-buffer approach relies on k being small so that it can be executed on shaders. For example, state-of-the-art volume renderers typically use two or six as the k value. Because these values are much smaller than actually required for correct visibility ordering, the visibility sorting may contain errors, which leads to visual artifacts.

The required k values varies significantly from dataset to dataset. Usually, the datasets with uniform, Delaunay-like tetrahedralizations, require small k values. Our

```
GPUCellProjectionAlgorithm();
begin
    CopyToGPU(Vertices);
    CopyToGPU(Tetrahedra);
    CopyToGPU(ColorMap);
    CopyToGPU(ViewParameters);
    GPU.Run(ComputeScreenSpaceProjections(Vertices));
    SSC = CopyFromGPU(ScreenSpaceCoordinates);
    IntersectionRecord *PerHashBlockIntersectionLists[NumOfRenderIterations];
    ExtractIntersectionRecords(Tetrahedra, Vertices, SSC,
    PerHashBlockIntersectionLists);
    for $i = 0$ upto NumOfRenderIterations do
        CopyToGPU(PerHashBlockIntersectionLists[$i$]);
        GPU.Run(processHashBlock);
        GPU.Run(sortHashBlock);
        GPU.Run(composeHashBlock);
    Display(OutputImage);
end
```

Algorithm 5: GPU-based cell-projection algorithm

focus is more on the quality of rendering rather than speed. We also want to support datasets with irregular tetrahedralizations without affecting visual accuracy. Accordingly, we do not use the k-buffer approach and implement a high accuracy visibility sorting mechanism.

4.1 CUDA implementation

Although the cell-projection algorithm is highly suitable for parallel implementation, to increase efficiency, hardware restrictions and capabilities must be considered. Graphics cards contain many processing units, but as each computation unit is much slower than a CPU core, their computation power depends on a high level of parallelism. Memory is another important restriction. Current GPUs usually have fewer than 2 GBs of memory, with 1 GB of memory more common. Since volume data can be very large, such memory restrictions can easily result in a bottleneck. The algorithm should consider such restrictions and be able to work with limited memory.

We present our GPU implementation of the cell-projection algorithm using CUDA in Algorithm 5. The serial version processes each pixel individually, but this approach is not suitable for GPUs. First of all, each pixel's workload is too low to be efficiently parallelized; the number of CUDA threads should be at least in the tens of thousands. The number of intersection records per pixel would be much less. Assigning one pixel's process to different threads, similar to multicore implementation, is also not suitable, as the execution flow and memory access patterns are unorganized. Further, each pixel's workload differs drastically, which would cause significant workload imbalance among threads.

For the GPU, we grouped the pixels, calling each group a hash block, and processed one group at each iteration. The *NumOfRenderIterations* variable represents the number of hash blocks, which depends on the dataset size. If the value of *NumOfRenderIterations* is low, the workload per render iteration increases, which allows more efficient parallelization, but increases the memory requirement. The amount of available memory thus limits the number of iterations. In our implementation, we used 16 or 64 rendering iterations. We describe the steps of the GPU-based algorithm below.

CPU to GPU data transfer The tetrahedra and vertices data are copied just once after CUDA initialization is completed. The view parameters are copied at the beginning of each rendering, i.e., whenever the volume is redrawn. The color map is copied at the beginning, but can be updated if the transfer function changes. We assume the graphics card contains enough memory to hold these data, leaving some space. The data transfer between the main memory and the GPU memory is very fast and has little effect on rendering times.

Screen space projection of vertices This function runs on the GPU using 512 blocks with 256 threads each. Usually using the same computation, each thread calculates the screen space coordinates of one vertex; the execution flow differs only if exceptions are observed, which is not frequent. Vertices are assigned to threads sequentially; thus memory access patterns are uniform and the shared memory is highly utilized. As a result, this function is very efficiently parallelized.

Extracting intersection records This step can be considered the screen space projection of the tetrahedra and is executed on the CPU. After the screen space coordinates of the vertices are calculated, they are copied back to the CPU. This operation takes little time, since the data size is small and the data transfer rates are high. This algorithm works similarly to its CPU version; the only difference is that instead of the intersection records being grouped according to their pixel coordinates, they are grouped according to their hash blocks. A pixel's hash block is simply computed as follows: The last two or three bits of a pixel's x and y coordinates are concatenated. With two bits, 16, and with three bits 64, hash blocks are addressed. The most important property of the hash function, which helps balance the workload of each hash block, is to ensure that the pixels in each block represent a subsampling of the whole image rather than being grouped in certain parts of the image.

We implemented this function on CPU because of irregularity and memory concerns. Each tetrahedron has a different projection area, and the execution flow of the scan-line algorithm differs for each tetrahedron. The number of pixels in each tetrahedron's projection area differs significantly, unevenly distributing the workload. Most importantly, race conditions will be observed. Many tetrahedra will have projections on the same hash blocks, thus they will try to write into identical locations. The only solution for such race conditions is to extract the intersection records first and organize them into hash blocks later. Although such an approach works, it introduces significant overhead.

Processing hash blocks Apart from GPU-specific optimizations, this step is similar to the calculation of intersection effects in the CPU version of the algorithm. It starts by copying the intersection records for the current hash block to the GPU. Each thread is assigned an intersection record and responsible for producing the corresponding intersection effect data. Execution flows are mostly uniform. Continuous threads are assigned to process contiguous intersection records; thus, memory accesses are also uniform for some parts. However, the tetrahedra contain references to vertices, which are not contiguous, and accesses to the vertices cannot be made uniform. However, for repeated access to nonuniform data, the algorithm uses shared memory to store the data temporarily, which makes the subsequent memory accesses uniform and much faster. This function is launched with a grid of 1024 blocks; because of the size difference of the shared memory between devices, those with a computing capability of 2.0 or higher are launched with 192 threads per block and others use 64 threads per block.

Sorting hash blocks This step for sorting intersection effects differs from the one in the CPU version of the algorithm significantly. In the CPU version, the intersection effects for each pixel must be sorted, and thus, many small sorting jobs are done independently, which enables efficient use of sorting algorithms like *quicksort*. However, in the CUDA version, intersection effects for every pixel in a hash block are mixed; the sorting function must group an individual pixel's intersection effects and then sort these groups. Further, in the CUDA version, the sort lists are much larger. In this work, we used efficient radix sort implementation by Satish et al. [14] from the Thrust library [6]. This library includes optimized implementations of various functions for GPUs and, integrated with CUDA, it can use device memory allocated from there. Radix sort has *O(n)* time complexity, which allows us to use larger hash blocks (thus fewer rendering iterations) without negatively affecting sorting times. It can also be efficiently parallelized.

Radix sorting is not a comparison-based sorting technique. It uses standard data types, given as input arrays, as the key and takes another array of standard data types as data. Using the keys, it sorts the keys and the data. Since radix sort cannot use custom structures, we divided the data inside the intersection effect structure into different arrays: *pix*, *dist* and *clr*. The *pix* array (pixel) does not exist in the CPU version, but because sorting is not done on a per-pixel basis in this version, this distinction is needed. The *pix* array elements are computed from the pixel coordinates, with the current rendering iteration revealing the last two or three bits of each pixel's x and y coordinates. The rest of the bits are packed and each *pix* value is computed. For example, let the resolution be 1024×1024 and the number of iterations be 16. Then each pixel's x and y coordinates can be represented with eight bits, making each *pix* value 16 bits in size.

Our sorting implementation must group the pixels' intersection effect data, then sort the groups according to distance. Since radix sort is stable, sorting first according to distance and then according to the *pix* value achieves this.

Algorithm 6 shows the sort algorithm. *index* is an empty buffer, and the algorithm begins by filling that buffer with the integer sequence $0, 1, 2, \dots,$ representing the data index. The data is sorted using *dist* as the key and *index* as the data. Then the

```
SortHashBlock(uint *pix, float *dist, uint *index, color *clr);
begin
    thrust.sequence(index);
    thrust.sort(dist, index);
    thrust.gather(index, pix);
    thrust.sort(pix, index);
    thrust.gather(index, clr);
end
```

Algorithm 6: GPU-based sort algorithm for hash blocks

```
ComposeHashBlock(int iteration, uint *pix, color *clr);
begin
    color currFB[$\frac{Width \times Height}{NumOfRenderIterations}$];
    Reset(currFB);
    while $length > CPUSwitchThreshold$ do
        GPU.Run(Reduce(pix,clr,currFB));
    $hostPix$ = CopyToCPU(pix);
    $hostClr$ = CopyToCPU(clr);
    $hostCurrFB$ = CopyToCPU(currFB);
    for $i = 1$ upto hostPix.length do
        composeRecord(hostCurrFB[hostPix[i]], hostClr[i],
        hostCurrFB[hostPix[i]]);
    $currFB$ = CopyToGPU(hostCurrFB);
    GPU.Run(updateFrameBuffer(currFB, iteration));
end
```

Algorithm 7: Composition algorithm for hash blocks

pix array must be reordered according to the new indices from the *index* array. The *gather* function from the *Thrust* library performs these reorderings.

The initial sorting step sorts the data according to eye distances. The second step, using *pix* as the key and *index* as the data, groups the data. With the final gather, the *pix* and *clr* arrays are sorted by eye distance and grouped by pixel values.

Composing hash blocks This step for composing the intersection effects uses the sorted *clr* and *pix* arrays as input. It combines the color values in sorted order for each pixel and computes the final pixel color. The composition process is described in Algorithms 7 and 8.

Algorithm 7 uses a temporary frame buffer, *currFB*. The size of this buffer is equal to the number of pixels processed at each iteration. Algorithm 7 runs on the CPU. It repeatedly calls the *Reduce* function (Algorithm 8), which runs on the GPU and reduces the size of the array by half. This function relies on the input arrays

```
Reduce(uint *$pix$, color *$clr$, color *$currFB$);
begin
    $tid$ = block.$id$ × block.$size$ + thread.$id$;
    $index = tid$;
    while $index < pix.length$ do
        if $pix[2 \times index] == pix[2 \times index + 1]$ then
            composeRecord($clr[pix[2 \times index]]$, $clr[pix[2 \times index + 1]]$,
            $clr[pix[2 \times index]]$);
        else
            composeRecord($currFB[pix[2 \times index + 1]]$, $clr[pix[2 \times index + 1]]$,
            $currFB[pix[2 \times index + 1]]$);
        $index$ += grid.$size$ × block.$size$;
end
```

Algorithm 8: Reduction algorithm for hash blocks

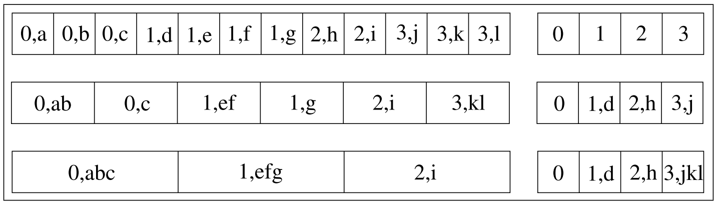


Fig. 1 Reduction example

being sorted, as each thread processes a consecutive pair of entries in the input arrays. First, the *pix* values of these pairs are compared. If the values are equal, then because the arrays have been sorted, the two color values can be combined and represented as a single color value. If the *pix* values are not equal, the later record is output and the earlier record is retained. A pair's *pix* values can be different only if the later record is the first record of the pixel that has not yet been output.

Figure 1 illustrates the algorithm with an example. The process starts with 12 *pix*, *clr* entries. The right-hand side shows the temporary frame buffer, which is initially empty. After the first reduction, the data shrinks to six entries. $(1, d)$, $(2, h)$, and $(3, j)$ entries are output to frame buffer while $(0, c)$, $(1, g)$, and $(2, i)$ entries are retained. Entry couples with the same *pix* values are combined to obtain $(0, ab)$, $(1, ef)$, and $(3, kl)$. Further reductions are executed in the same way. To reduce two entries to one, the algorithm uses the *composeRecord* function, which takes three colors as input. It combines its first two inputs and writes the result to its third input. This function works similarly to the serial version.

The result of the *Reduce* function is an interleaved array. Further reductions can be performed on the interleaved arrays, or the array can be compacted. We use an optimized compaction mechanism. After the first reduction, we compact the interleaved array into its first half using the gather functionality of the *Thrust* library. This operation frees the other half for temporary storage. Further reduction operations write their results to that free space. This approach eliminates the need to compact reduction operations beyond the first one and allows parallelization of memory accesses for greater efficiency.

In the *Reduce* function, each thread is responsible for combining two entries into one. With further reductions, the arrays shrink significantly; thus after a certain point, the threads cannot be utilized efficiently. We used two mechanisms to overcome this problem. (i) We run the *Reduce* function with size-dependent grid and block sizes. With large sizes, we use 512 blocks per grid and 256 threads per block. With smaller sizes, we stepwise reduce the size to 64 blocks per grid and 32 threads per block. (ii) After a certain length, we complete the reductions in the CPU. Because the data is now so small, transfers between the CPU and GPU take little time and the CPU executes the reduction operations more efficiently because parallelism is limited.

After the reductions are complete, all intersection effects are combined into the temporary frame buffer. With the *updateFrameBuffer* function, which runs on the GPU, the data in this buffer is transferred into the actual frame buffer. The naive version completes the image after all render iterations have been completed; however, this function also supports progressive rendering, which is very useful for increasing interactivity.

4.2 Progressive rendering

Volume rendering is time consuming, particularly for high resolutions, and this adversely affects the interactivity. Progressive rendering aims to perform a low-resolution volume rendering and progressively improve image quality while displaying the outputs to the user. As low-resolution rendering is faster, a low-quality image is displayed fairly quickly. As the process continues, the image progressively improves, while the user is observing the latest output.

The hash-block approach provides a natural framework for progressive rendering. As mentioned earlier, each hash block will produce the overall subsampling of the whole image. For example, if we use 16 hash blocks to render a 1024×1024 image, then each hash block will produce a 256×256 image, which is a low resolution version of the whole image. Each of those low resolution images are slightly shifted from each other, so that when combined they will constitute the high resolution image. Progressive rendering simply requires processing the hash blocks in a specific order, so that the processed hash blocks can be combined to obtain higher and higher resolutions progressively.

The pixel values are assigned to each hash block according to their last two or three bits. This approach divides the whole image into 4×4 or 8×8 sub-windows, respectively. Each pixel in a subwindow is processed by a different hash block. Without progressive rendering, only the values of pixels assigned to the currently processed hash block are updated in the frame buffer. On the other hand, progressive rendering requires some of the neighboring pixels being updated as well. For example, after the first hash block has been processed, every pixels' values in a subwindow is updated with the computed pixel's value. Accordingly, the lower resolution version can be displayed without waiting the rendering to finish.

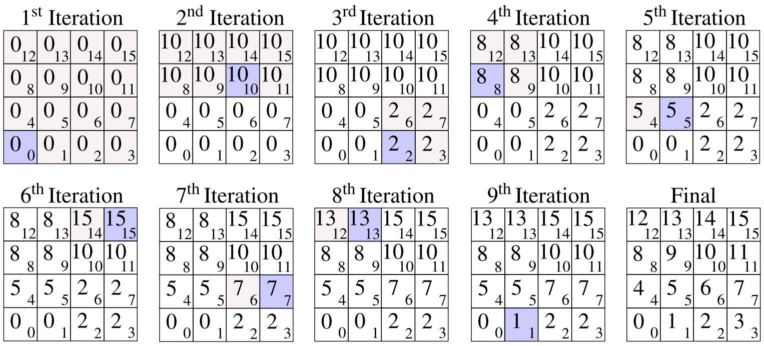


Fig. 2 Progressive rendering

Figure 2 illustrates the progressive rendering process. In the example, we have 16 render iterations and a 4×4 subwindow. Right bottom corner of each cell displays the index value of the pixel within the subwindow. The larger value represents the index of the pixel whose value is currently set to the current pixel. The blue background indicates the currently processed pixel, and the red background indicates the pixels whose values are updated with current pixels values.

In the first iteration, the 0th pixel is processed and the results are written to every pixels. In the second iteration, the 10th pixel is processed and the results are written to pixels on the top half. At each iteration another pixel is processed and the results are written to some neighboring pixels, until every pixel value is computed.

The processing order of pixels is important for progressive rendering. With the given ordering, we can obtain various sub-resolutions after certain rendering iterations. For example, we can output, 1×1, 2×1, 2×2, 4×2, and 4×4 sub-resolutions of our 4×4 subwindows after 1st, 2nd, 4th, 8th, and 16th iterations respectively. As a result, the low resolution versions can be quickly obtained and displayed, while the resolution progressively improves. This technique has very little overhead, but improves the interactivity greatly.

4.3 Memory management

Memory is usually the limiting factor for the size of volume data that a renderer can visualize. We have employed several methods to keep both the GPU and system memory requirement low. Our motivation was to keep the memory footprint as low as possible without adversely affecting the performance or accuracy.

Although system memory is large, it can become a limiting factor for high-image resolutions, especially because the cell-projection algorithm extracts the intersection records and stores them in the memory before processing. To work with the available memory, our implementation can render images in multiple passes. For example, an image with 4096×4096 resolution can be rendered in four passes of 2048×2048 or 16 passes of 1024×1024. This approach introduces some overhead; however, it guarantees that the application will fit in the physical memory with no swapping, and thus improves performance. GPU memory requirement is more crucial because it is smaller. Vertex and tetrahedron data are the main components of the volume data and should be placed in the GPU memory.

Tetrahedron structure contains indices of four vertices, which comprise the tetrahedron. Each index value is a decimal value between zero and the number of vertices in the dataset. Thus $\lceil \log_2(\mathit{NumberOfVertices})\rceil$ bits are enough to represent an index value. Accordingly, the minimum number of bits that is sufficient to represent a tetrahedron is $4 \times \lceil \log_2(\mathit{NumberOfVertices})\rceil$ bits. For example, tetrahedra in a volume with half a million vertices can be represented with 76 bits.

CUDA requires uniform accesses to memory in order to give high performance. For example, if all the threads in a CUDA warp accesses consecutive 32 bit data at a certain memory region, all memory operations can be completed concurrently. However, if the accessed data types are not 32 bits or the accessed values are not consecutive, more memory accesses will be required. Thus, using a multiple of 32 bits to represent a tetrahedron is crucial for a uniform memory access. Although we used 128 bits (16 bytes), 96 bits (12 bytes) is sufficient to represent a reasonably-sized volume data and it is a multiple of 32 bits. However, index values must be encoded for 12 bytes whereas using 16 bytes do not require any encoding. Although decoding encoded index values require simple bitwise arithmetic, it still introduces some computational overhead for encoding and decoding. We experimented encoding into 12 bytes to reduce the memory size, but we observed noticeably higher rendering times. The memory requirement for storing vertices is minimal. The total memory requirement for storing volume data is $16 \times (\mathit{NumberOfVertices} + \mathit{NumberOfTetrahedra})$. For the largest dataset, we tested sf1 with 14 millions of tetrahedra; tetrahedra and vertices require about 250 Mbs of GPU memory.

The maximum possible number of intersection records in a hash block is directly proportional to the memory requirements during the hash block processing. This value depends on the dataset and view parameters greatly. On the other hand, this value can be easily reduced by using higher number of hash blocks or using multi-pass rendering. Accordingly we can change the memory requirement according to the available memory. In our implementation, 34 bytes per intersection record is needed. This memory is reused as much as possible during the processing of hash blocks. When the sf1 dataset is rendered using 64 hash blocks with a 1024×1024 resolution in a single pass, 120 Mbs of memory is needed for hash block processing. Together with tetrahedra, vertices, and other smaller data structures, the memory requirement falls well below 500 MBs. Accordingly, a graphics card with 2 GBs of memory would be sufficient to render a volume of a hundred millions of tetrahedra with similar characteristics to sf1 dataset using 64 hash blocks with a 2048×2048 resolution in a four passes.

5 Results

We used a PC with a four-core AMD CPU running at 3.2 GHz, 4 GB of system memory, and an Nvidia GTX 560 graphics card with 1 GB of memory and 1075 GFLOPS

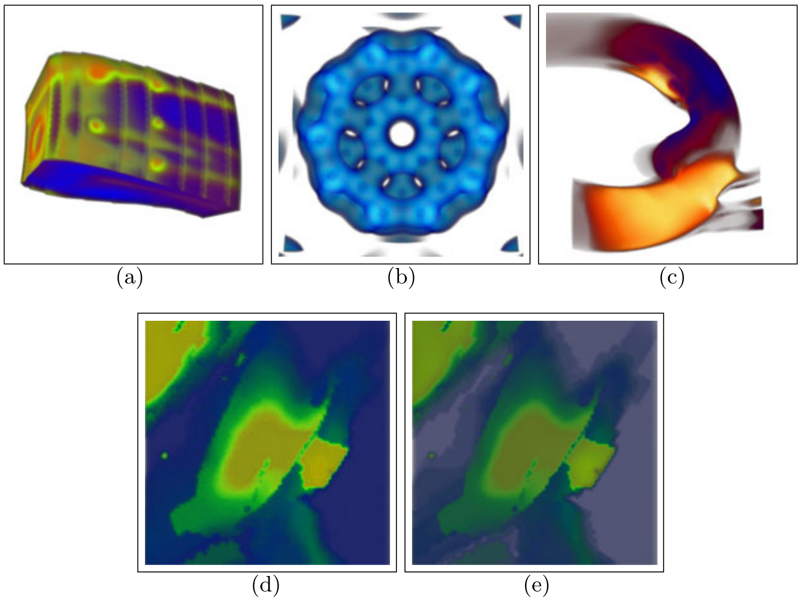


Fig. 3 Rendered images of various datasets: (**a**) Comb dataset, (**b**) Bucky dataset, (**c**) Aorta dataset, (**d**) Sf2 dataset, and (**e**) Sf1 dataset

single precision arithmetic capabilities. We tested our implementations on five different datasets (see Fig. 3). We rendered each dataset with three different view parameters and for three different resolutions: 512 × 512, 1024 × 1024, and 2048 × 2048. We report the average results for each resolution.

Table 1 shows that significant speed-ups are obtained with the multicore and GPU implementations. The multicore implementation achieves a 3.2-fold increase in speed on average. Considering the rendering has a significant input-output (IO) component, these speed-ups are very promising. The GPU implementation achieves 23.3-fold increase in speed on average, with some speed-ups reaching above 32-fold. Considering we used a middle-segment graphics card in the tests, these speed-ups are also very promising.

Figure 4(a) shows the speed-ups obtained for various resolutions of different datasets using the multicore implementation. This implementation achieves almost identical speed-ups for 512 × 512 and 1024 × 1024 resolutions. For the 2048 × 2048 resolution, speed-ups are slightly slower because of memory limitations. Our multi-pass rendering approach solves high-memory requirement problem, but introduces some overhead, which reduces the speed. For resolutions above 2048 × 2048, we expect speed-ups to be higher because serial implementation will also incur multi-pass rendering overheads.

The GPU implementation produces higher speed-ups for the 1024 × 1024 and 2048 × 2048 resolutions (cf. Fig. 4(b)) because, larger jobs use the GPU's tens of thousands threads more efficiently. The 512 × 512 resolution does not utilize the graphics hardware as much, resulting in lower speedups. The multipass rendering overhead affected the speed-up of 2048 × 2048 resolution, but particularly for smaller

Table 1 Rendering times and speed-ups of GPU, multicore, and serial cell-projection algorithms. Data size is given in thousands of tetrahedra. Rendering times are in seconds

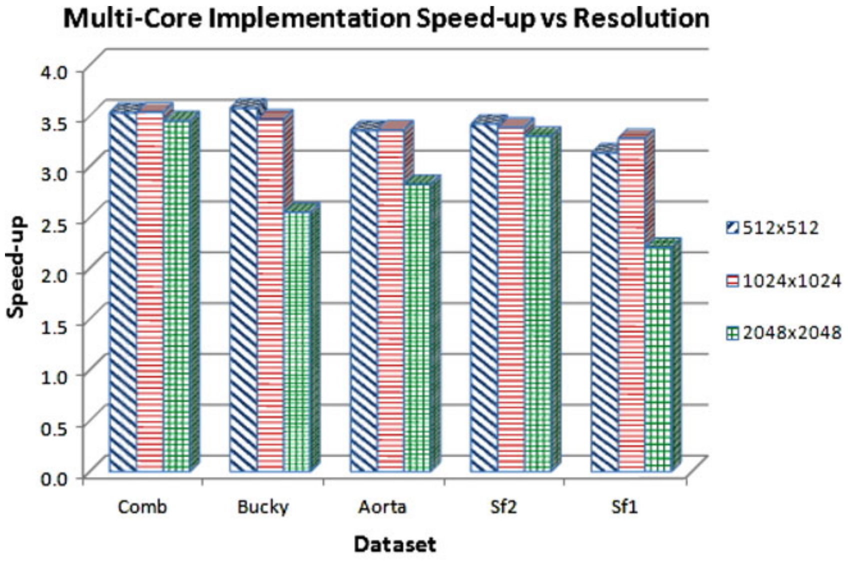
Dataset	Data size	Resolution	Serial	Multi-core		GPU	
			Time	Time	Speed-up	Time	Speed-up
Comb	215.0	512×512	10.45	2.96	3.531	0.53	19.863
		1024×1024	41.64	11.77	3.537	1.61	25.877
		2048×2048	168.19	48.72	3.452	5.97	28.179
Bucky	1250.2	512×512	71.76	20.07	3.576	3.16	22.735
		1024×1024	285.94	82.45	3.468	10.33	27.671
		2048×2048	1334.23	523.23	2.550	47.57	28.049
Aorta	1386.9	512×512	31.70	9.45	3.356	1.27	25.055
		1024×1024	125.06	37.21	3.361	4.09	30.547
		2048×2048	554.41	196.12	2.827	16.99	32.633
Sf2	2067.7	512×512	40.34	11.80	3.418	2.44	16.516
		1024×1024	159.48	47.08	3.387	6.77	23.554
		2048×2048	637.79	193.30	3.300	28.76	22.180
Sf1	13980.1	512×512	82.77	26.41	3.135	6.63	12.484
		1024×1024	318.24	97.04	3.280	18.19	17.492
		2048×2048	1615.24	732.10	2.206	100.01	16.152

datasets, the associated overhead was balanced by the higher utilization. For resolutions above 2048×2048, we expect that speed-ups would remain high.

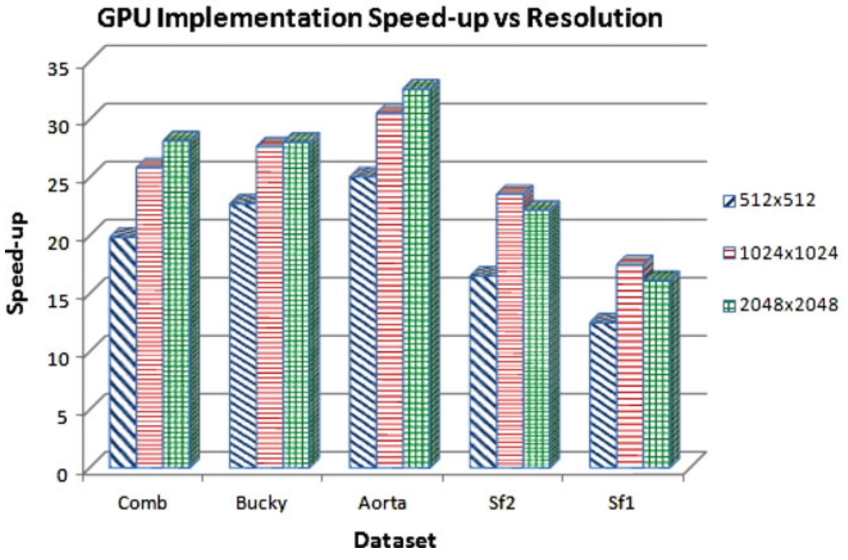
6 Conclusions

We propose multicore and GPU implementations of the cell-projection algorithm for direct volume rendering. The proposed algorithms are designed in such a way that they use the main memory and the GPU memory efficiently. With the multi-pass rendering approach, our implementation is able to render high resolution images. We tested our implementations with several large datasets with different characteristics and under various view parameters. The multicore implementation produced up to 3.5-fold speed-ups, while the GPU implementation reached 32-fold speed-ups. Together with the progressive rendering mechanism, we achieved interactive rates for many datasets.

Acknowledgements The authors are grateful to Koji Koyamada for the volumetric datasets. The Comb dataset is courtesy of NASA. The Sf1 and Sf2 datasets are courtesy of David R. O'Hallaron and Jonathan R. Shewchuk (CMU). We are grateful to Rana Nelson for proofreading.



(a)



(b)

Fig. 4 Speed-ups for various resolutions of different datasets: (**a**) multi-core implementation and (**b**) GPU implementation

References

1. Berk H, Aykanat C, Güdükbay U (2003) Direct volume rendering of unstructured grids. Comput Graph 27(3):387–406

2. Callahan SP, Ikits M, Comba JLD, Silva CT (2005) Hardware-assisted visibility sorting for unstructured volume rendering. IEEE Trans Vis Comput Graph 11(3):285–295
3. Cook R, Max NL, Silva CT, Williams PL (2004) Image-space visibility ordering for cell projection volume rendering of unstructured data. IEEE Trans Vis Comput Graph 10(6):695–707
4. Engel K, Kraus M, Ertl T (2001) High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware, HWWS'01. ACM, New York, pp 9–16
5. Garrity MP (1990) Raytracing irregular volume data. In: Proceedings of the IEEE workshop on volume visualization. ACM, New York, pp 35–40
6. Hoberock J, Bell N: (2012) Thrust: a parallel template library. Version 1.3.0. http://www.meganewtons.com
7. Koyamada K (1992) Fast traversal of irregular volumes. In: Visual computing—integrating computer graphics with computer vision. Springer, Berlin, pp 295–312
8. Kraus M, Ertl T (2001) Cell-projection of cyclic meshes. In: Proceedings of IEEE visualization, pp 215–559
9. Lefohn AE, Kniss JM, Hansen CD, Whitaker RT (2004) A streaming narrow-band algorithm: interactive computation and visualization of level sets. IEEE Trans Vis Comput Graph 10:422–433
10. Lorensen WE, Cline HE (1987) Marching cubes: a high resolution 3D surface construction algorithm. Comput Graph 21(4):163–169
11. Okuyan E, Güdükbay U, İşler V (2012) Dynamic view-dependent visualization of unstructured tetrahedral volumetric meshes. J Vis 15:167–178. doi:10.1007/s12650-011-0122-x
12. Rezk-Salama C, Engel K, Bauer M, Greiner G, Ertl T (2000) Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware, HWWS'00, ACM, New York, pp 109–118
13. Roettger S, Guthe S, Weiskopf D, Ertl T, Strasser W (2003) Smart hardware-accelerated volume rendering. In: Proceedings of the symposium on data visualisation, VISSYM'03. Eurographics Association, Aire-la-Ville, pp 231–238
14. Satish N, Harris M, Garland M (2008) Designing efficient sorting algorithms for manycore GPUs. Tech Rep NVR-2008-001, NVIDIA Corporation
15. Shirley P, Tuchman A (1990) A polygonal approximation to direct scalar volume rendering. ACM SIGGRAPH Comput Graph 24(5):63–70
16. Silva CT, Comba JLD, Callahan SP, Bernardon FF (2005) A survey of GPU-based volume rendering of unstructured grids. Rev Inform Teõr Apl 12(2):9–30
17. Weiler M, Kraus M, Merz M, Ertl T (2003) Hardware-based ray casting for tetrahedral meshes. In: Proceedings of the 14th IEEE visualization, VIS'03. IEEE Comput Soc, Los Alamitos, p 44
18. Wylie B, Moreland K, Fisk LA, Crossno P (2002) Tetrahedral projection using vertex shaders. In: Proceedings of the IEEE symposium on volume visualization and graphics, VVS'02. IEEE Press, Piscataway, pp 7–12