DOI: 10.1111/j.1467-8659.2009.01674.x

Eurographics/ IEEE-VGTC Symposium on Visualization 2010
G. Melançon, T. Munzner, and D. Weiskopf
(Guest Editors)

Volume 29 (2010), Number 3

Multi-layer Depth Peeling by Single-Pass Rasterisation for Faster Isosurface Raytracing on GPUs

Baoquan Liu, Gordon J. Clapworthy and Feng Dong

Department of Computer Science and Technology, University of Bedfordshire, UK

Abstract

Empty-space skipping is an essential acceleration technique for volume rendering. Image-order empty-space skipping is not well suited to GPU implementation, since it must perform checks on, essentially, a per-sample basis, as in kd-tree traversal, which can lead to a great deal of divergent branching at runtime, which is very expensive in a modern GPU pipeline. In contrast, object-order empty-space skipping is extremely fast on a GPU and has negligible overheads compared with approaches without empty-space skipping, since it employs the hardware unit for rasterisation. However, previous object-order algorithms have been able to skip only exterior empty space and not the interior empty space that lies inside or between volume objects.

In this paper, we address these issues by proposing a multi-layer depth-peeling approach that can obtain all of the depth layers of the tight-fitting bounding geometry of the isosurface by a single rasterising pass. The maximum count of layers peeled by our approach can be up to thousands, while maintaining 32-bit float-point accuracy, which was not possible previously. By raytracing only the valid ray segments between each consecutive pair of depth layers, we can skip both the interior and exterior empty space efficiently.

In comparisons with 3 state-of-the-art GPU isosurface rendering algorithms, this technique achieved much faster rendering across a variety of data sets.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing Algorithms

1. Introduction

Isosurface rendering is a simple and effective approach for visualising distinct objects present in the data. It displays a surface representing the locus of a collection of points in the volume that correspond to a given isovalue (i.e. greyscale intensity in the data). Isosurface rendering is used by domain experts, such as doctors, to visualise the exact boundaries of materials with complex shapes, such as skin, skull or perhaps a tumour. It must be accurate, as they are frequently interested in viewing details in close up. It must also be fast because, when they are investigating the data and seeking to create the precise isosurface they require, the interaction will demand that frequent changes of the isovalue are made at runtime. In this context, methods that employ costly precomputation to build complex acceleration structures, such as kd-trees [VMD08], which become invalid and need to be rebuilt whenever the isovalue is changed, are generally less efficient than methods that perform calculations on the fly.

In this paper, we accelerate GPU based isosurface raytracing by applying multi-layer depth peeling using only a single hardware rasterising pass, during which we determine all of the depth layers of the tight-fitting bounding geometry of the isosurfaces. The maximum number of layers peeled by our approach can be in the thousands with the depth of each layer recovered in 32-bit float-point format (fp32), which is sufficiently accurate to cover all the valid ray segments and ensure that no details of the isosurfaces are missed in the subsequent raytracing.

Our major contribution is that (to our knowledge) this is the first method that makes it possible to peel thousands of depth layers by a single hardware rasterisation pass (comparing with the existing methods that can peel at most 32 layers per pass). The resulting layers are used to accelerate GPU raycasting to a degree that has not been possible previously. More detailed contributions are given below:

1. We employ the hardwired rasterisation unit (a special-

purpose fixed-function component on the GPU) to rasterise the active cells (i.e. those through which the isosurface passes) and generate bitmasks to indicate the boundary layers.
2. We accurately reconstruct all of the depth layers (in fp32 format) from the bitmasks. The resulting layers generate depth intervals bordered by two consecutive depth layers, which define all of the valid ray segments. By tracing rays only within these ray segments, we can efficiently skip both the interior and exterior empty space.
3. To decrease the number of the ray segments, we merge runs of set bits in a bitmask into a bit-segment (with two borders) using a texture lookup, which is much more efficient than performing bit counting in the fragment shader.
4. Our method is well suited for transparent rendering of multiple isosurface-layers as all of the generated valid ray segments are in front-to-back order, so accurate transparent blending is straightforward.
5. Array texture is used for the first time for depth peeling, and hence our output per fragment is very slim (only two bytes), thus avoiding profligacy with memory bandwidth, unlike previous approaches that have used extremely fat frame buffers, i.e. multiple rendering target (MRT), and hence need to output 128 bytes per fragment.

2. Related Work

2.1. Empty space skipping

Empty-space skipping is an essential acceleration technique for volume rendering, which typically employs either an image-order or an object-order approach (we refer the reader to [HLSR09] for a comprehensive survey).

Vidal et al. [VMD08] used kd-trees, built on the CPU in a preprocessing stage, to remove empty space. This stage consumes about 4 seconds for 512^3 data sets and must be repeated whenever the transfer function or isovalue is changed. As a result, this approach is not suitable for dynamic data exploration. Recently, Hughes et al. [HL09] implemented Kd-Jump, a GPU-based kd-tree scheme for isosurface raytracing. A change in isovalue requires an update of every node of their kd-tree at every level, starting from the original volume itself; this takes 0.25 seconds for 512^3 volumes with a CUDA implementation.

Many previous researchers have sought to shorten the ray length by applying object-order empty-space skipping to improve GPU raycasting performance [KW03, HLSR09, LCD09a]. However these algorithms generally skip only exterior, but not interior, empty space. The work in [LCD09b] skipped some interior empty space but it ray-traced each active cell independently of the others. As a result, it produces many overlapping ray segments, which are redundantly ray-traced over and over again by different cells. Polygon Assisted Ray Casting (PARC) was proposed in [SA95] to accelerate volume raytracing, where the rectangular polygons were rasterized so that the CPU-based cell-by-cell stepping algorithm can avoid evaluating empty cells.

2.2. Depth peeling

The use of depth peeling [Eve01] makes it possible to handle complex geometries with many depth layers. In the context of GPU-based volume rendering, depth peeling has been used on the bounding geometries of multi-volumes to generate all of the individual depth layers for CSG operations [RBE08]. Unfortunately, depth peeling requires $O(N)$ geometry rasterisation passes and $O(N)$ depth comparisons per pass, leading to a total complexity $O(N^2)$, where N is the depth complexity (the total number of layers) of the geometries. The N passes of rasterisation makes this approach unsuitable for real-time applications with complex geometries.

Peeling multiple layers by a single GPU rasterisation pass was first proposed by Liu et al. [LWX06]. Other algorithms followed, including K-buffer [BCL*07], stencil routed A-buffer [MB07] and dual depth peeling [BM08], but all of these can peel at most 8 layers per pass. Recently, Liu et al. [LHLW09a] made it possible to peel up to 32 layers per pass, but this method always produces errors when any two layers are located close together. This potential error is acceptable for approximate rendering, as needed for games, but not for medical applications, which require that no detail of the geometry should be missed. Another drawback is that this method solves the multi-pass issue by profligate use of bandwidth as it uses extremely fat frame buffers (MRT), with 128 bytes having to be output for each fragment.

Recently Liu et al. [LHLW09b] proposed single-pass depth peeling via a CUDA rasteriser. This leaves the hardware rasterising unit idling, and instead uses a software rasteriser, which makes extensive use of atomic operations in global memory. It is also necessary to sort the rasterised fragments explicitly, which could become a bottleneck when the depth complexity of the geometry is high. The maximum layer count tested in their experiments was only 15.

Recent GPU-based binary voxelisation algorithms [ED06, ED08] provide better performance than depth peeling, but they derive only a binary boundary voxelisation. This works well for approximating volumetric effects, but its binary representation of each layer is insufficient for applications that need full fp32 precision for each depth value, which depth peeling can accomplish gracefully. Another drawback is their use of MRT which, as mentioned previously, means that 128 bytes must be output per rasterised fragment.

3. Algorithm Overview and Flowchart

In the context of isosurfacing, volume datasets tend to be rather sparse, with many inactive voxels; these occupy much of the total volume but make no contribution to the isosurface. Previous object-order algorithms [HLSR09, LCD09a] have used only two layers (the front-most and back-most faces) of some bounding geometries to set up rays, which is normally too coarse. Also, they can only skip exterior empty space but not the interior empty space lying within or between multiple volume objects, as shown in Figure 1. Our method overcomes these problems by rasterising the active cells and generating an accurate series of valid ray segments, and it is only along these that GPU raycasting is performed.

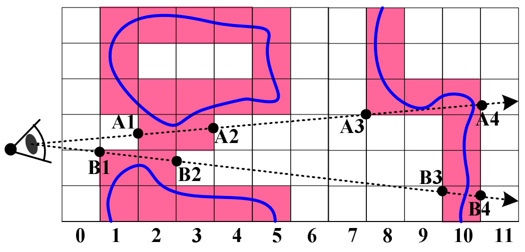

Figure 1: *The active cells (red) enclose some isosurfaces (blue). For the upper ray, A, our approach generates 2 valid ray segments in near-to-far order: $\overrightarrow{A_1A_2}$ and $\overrightarrow{A_3A_4}$; while for the lower ray, B, our approach generates 2 valid ray segments in the same order: $\overrightarrow{B_1B_2}$ and $\overrightarrow{B_3B_4}$. Previous object-order algorithms, which used only the first-hit and last-hit points, generated one coarse ray segment per ray, that is $\overrightarrow{A_1A_4}$ and $\overrightarrow{B_1B_4}$ for rays A and B, respectively, as a result they cannot skip interior empty space, such as $\overrightarrow{A_2A_3}$ and $\overrightarrow{B_2B_3}$.*

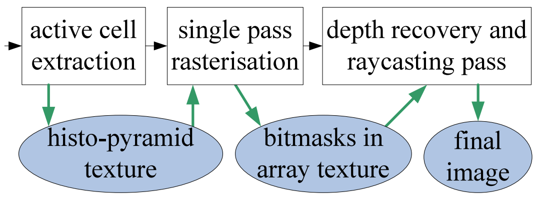
active cell extraction
single pass rasterisation
depth recovery and raycasting pass
histo-pyramid texture
bitmasks in array texture
final image

Figure 2: *A schematic view of the algorithm. Black arrows designate control flow, and green arrows designate data flow with the ellipses standing for the textures or images in video memory.*

Figure 2 shows a flowchart of our algorithm, which involves three stages. In the first stage, as soon as the isovalue is changed, we identify and extract all of the active cells on the GPU. In the second stage, we rasterise only these extracted cells as point primitives; for each fragment generated, we output a bitmask indicating the position of the active cell. The resulting bitmasks are routed into an array texture, which is used in the third stage, a full screen GPGPU pass, to recover all of the depth layers (in fp32 format) and cast rays only for the valid ray segments bordered by consecutive depth layers. In this way, we can skip both the interior and exterior empty space efficiently.

4. Implementation Details

In the following subsections, we shall provide greater detail about the 3 individual stages.

4.1. Stage 1 - active cell extraction

We define active cells as in the Marching Cubes (MC) algorithm [LC87]. From a 3D grid of $N \times M \times L$ scalar-value voxels, we form a grid of $(N-1) \times (M-1) \times (L-1)$ cube-shaped MC cells between the scalar values such that each corner of the cube corresponds to a scalar-value voxel. We maintain a regular grid of min-max values of enclosed voxels of each cell, which will be used to determine whether a cell is active or not by testing if the isovalue is within the range defined by the min-max pair of the cell.

Whenever the isovalue changes, we build a HistoPyramid texture [ZTTS06] in order to extract all of the active cells and skip all of the inactive cells, as in [LCD09b].

4.2. Stage 2 - rasterisation pass

Most previous algorithms [SA95,HQK05,HLSR09] handled each cell as a cube with up to 12 triangle primitives and rasterised the boundary faces of the cubes in several rendering passes. In these cube-based approaches, rasterising a tightly fitting bounding geometry involves many more vertex and geometry operations than for the point-based primitive used in [LCD09b], which handles only one primitive per cell with only one vertex operation and no need for primitive assembly or vertex connection. We thus employ this latter technique to rasterise each active cell as a point primitive, which leads to a round disk in the screen covering all of the pixels on to which the cell can project, e.g. the projection of the two cells shown in Figure 4.

Our active cell extraction and projection approaches are similar to [LCD09b], with the significant difference that, for each rasterised fragment, we output a 16-bit bitmask into a layered frame buffer (array texture) by using the geometry shader to route the projection of a cell into a certain layer of the array texture according to the position of the cell. This is known as layered rendering, in which a 2D array texture (essentially a 3D texture consisting of a stack of 2D texture slices) is bound to a frame buffer object. In contrast to the use of MRTs, in which each fragment generated by OpenGL outputs multiple copies of RGBA, layered rendering provides the freedom to route the output into different image layers of the array texture at a geometry shader level by setting *gl_Layer*. This technique is often used for dynamic Environment Mapping, whereby the six faces of the Cube-Map can be produced in a single rasterisation pass by using layered rendering. Here, for the first time, we use this technique for depth peeling.

To design a slim frame buffer (GL_R16UI, that is, a 16-bit single-channel frame buffer) by using an array texture, we need to divide the range (in the major axis direction) of the cell-slices into slabs. Each slab corresponds to a layer of the array texture and accommodates 16 slices of cells.

The division is shown in Figure 3. The major axis $\overrightarrow{axis_m}$

is selected according to which component of the optical axis vector $\overrightarrow{axis_{cam}}$ (of the camera) has the largest absolute value. The sign of the optical axis $sign_{oa}$ is set to 1 if this component is positive, otherwise it is set to -1. $TotalSlice$ and $TotalSlab$ are the total cell-slice number and total slab number, respectively, in the major axis direction. These simple variables are all calculated per frame on the CPU and transferred to the GPU shaders at runtime.

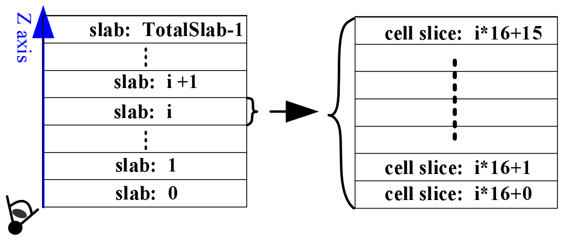

Figure 3: *Suppose the camera points in the Z-axis direction and the total slice number (TotalSlice) of cells is L-1, which can be divided into a series of slabs with each slab containing 16 slices of cells, then* $TotalSlab = \lceil L/16.0 \rceil$.

For each active cell, our vertex shader traverses the HistoPyramid to retrieve the object-space coordinates of the cell. Our geometry shader accepts a point primitive (an active cell) as input and outputs another point primitive with a different size and centre; these are calculated as the size and centre of the cell's projection (a round disk in screen-space) using the equations from [LCD09b]. We then set gl_Layer according to the sequence number, seq_{cell}, of the current cell along the major axis. Finally we calculate a bitmask as output for the cell, with one set bit indicating the position of the cell within the slab. We calculate the sequence number for each cell as $seq_{cell} = \lfloor key\_coor/sideLength \rfloor$, where $sideLength$ and key_coor are, respectively, the side length and component coordinate of the cell along the major axis.

If the camera points in the opposite direction to the major axis, that is, $sign_{oa} = -1$, we need to flip this sequence number: $seq_{cell} = TotalSlice - 1 - seq_{cell}$ in order to retain the near-to-far ordering relative to the camera position. The slab number into which the cell falls, $slab_{cell}$, and the relative position of the cell inside the slab, $bitPos$, can be calculated as: $slab_{cell} = \lfloor seq_{cell}/16 \rfloor$ and $bitPos = (seq_{cell}\%16)$. The bitmask for the cell can then be calculated as $bitmask = (1 << bitPos)$. An example of the calculation is shown in Figure 4.

In this way, our geometry shader sets $gl\_Layer = slab_{cell}$ to route the projection of the cell to a certain layer of the array texture; all rasterised fragments of the cell will output the same $bitmask$ in 16-bit integer format. Our output to global memory per fragment is thus very slim (only two bytes) compared with the previous fat frame buffer algorithms [ED06, ED08, LHLW09a], which output 128 bytes to global memory per fragment (for frame buffers of 8 MRTs, each having 4 channels of 32 bits). In the modern, many-core GPU architecture, memory bandwidth is one of the most important gating factors for performance, so this could provide a huge saving.

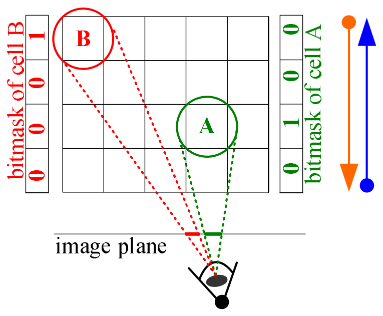

Figure 4: *If the camera is aligned with the major axis (blue arrow), then* Seq_{cell} *for cells A and B are 1 and 3, respectively; while if the camera direction is in the negative direction of the major axis (orange arrow), then* Seq_{cell} *for cells A and B are 2 and 0, respectively, before flipping, while they are 1 and 3 after flipping. So finally the bitmasks for cells A and B are 0x2 and 0x8, respectively. Here* $TotalSlice$ *is 4.*

By using OpenGL's logic operation (GL_OR), all incoming fragments will be combined into the array texture, and each generated fragment will set one bit in a certain layer of the array texture according to the position of the active cell.

4.3. Stage 3 - depth recovery and raycasting pass

In this full-screen GPGPU pass, we reconstruct the depth value (in fp32 format) from the bitmasks in the array texture and generate depth intervals, which define a series of valid ray segments; we then raytrace only inside these ray segments to render the isosurfaces. For clarity, we provide an overview of the fragment shader of this pass in Algorithm 1.

4.3.1. Depth recovery

Since bitmasks may contain some runs of set bits (i.e. a series of neighbouring 1s), instead of reconstructing depth values for all of the set bits, we can merge them into segments - we then need only to reconstruct depths for the two border bits of each segment.

This bit merging can be accomplished by a simple lookup table, which is much more efficient than performing bit counting in the fragment shader. We pre-compute, on the CPU, a 1D texture $maskTable$ with $2^{16} = 65536$ entries. Given a bitmask (a 16-bit unsigned integer) as an entry to look up the texture, it will return a texel with 3 channels (R, G and B, in 32-bit integer format): R gives the number of set-bit segments in the bitmask, while G and B respectively

Algorithm 1 DepthRecoveryAndRaycasting

```
1: $\overrightarrow{rayDir} \Leftarrow CalculateRayDirForThePixel(\overrightarrow{camPos}, gl\_TexCoord[0].xy)$;
2: $cosAphla \Leftarrow DotProduct(\overrightarrow{rayDir}, \overrightarrow{axis_m})$; //This could be positive or negative
3: $depth0 \Leftarrow (voxelSlice0 - camPos\_z)/cosAphla$;
4: $delta_{depth} \Leftarrow sideLength/cosAphla$;
5: for $iSlab = 0$ to $TotalSlab - 1$ do // iterating on all slabs
6:    $bitmask \Leftarrow arrayTextureLookup(array\_texture, gl\_TexCoord[0].xy, iSlab)$;
7:    $cellsOfPreviousSlabs \Leftarrow iSlab * 16$;
8:    $bit\_segments \Leftarrow TextureLookup1D(maskTable, bitmask).rgb$;
9:    for $iSeg = 0$ to $bit\_segments.r - 1$ do // iterating on all segments
10:       $start_{index} \Leftarrow cellsOfPreviousSlabs + RetrieveIndex(bit\_segments.g, iSeg)$;
11:       $end_{index} \Leftarrow cellsOfPreviousSlabs + RetrieveIndex(bit\_segments.b, iSeg)$;
12:       $startVoxel_{index} \Leftarrow FlipAndOffsetStartIndex(start_{index}, sign_{oa})$;
13:       $endVoxel_{index} \Leftarrow FlipAndOffsetEndIndex(end_{index}, sign_{oa})$;
14:       $start_{depth} \Leftarrow abs(depth0 + startVoxel_{index} * delta_{depth})$;
15:       $end_{depth} \Leftarrow abs(depth0 + endVoxel_{index} * delta_{depth})$;
16:       $\overrightarrow{start_{pos}} \Leftarrow \overrightarrow{camPos} + start_{depth} * \overrightarrow{rayDir}$;
17:       $\overrightarrow{end_{pos}} \Leftarrow \overrightarrow{camPos} + end_{depth} * \overrightarrow{rayDir}$;
18:       $[bFinding, Intersection] \Leftarrow RaytracingOneSegment(isovalue, \overrightarrow{start_{pos}}, \overrightarrow{end_{pos}})$;
19:       if $bFinding = ture$ then
20:          Shading($Intersection$); //shading the first intersection,
21:          return; //and then exit
22:       end if
23:    end for
24: end for
```

encode the start and end indices of each segment: $start_{index}$ and end_{index}.

For example, in Figure 1 the bitmasks for rays A and B are "011100001100" and "010000000110", respectively (where the least significant bit is to the right). Both bitmasks have 2 set-bit segments, with $[start_{index}, end_{index}]$ in pairs: {[2,3], [8,10]} for ray A and {[1,2], [10,10]} for ray B. This information can be retrieved when a bitmask is used as an entry to look up the 1D texture (lines 10-11 of Algorithm 1). Since these indices are the relative position inside a slab (as shown on the right of Figure 3), we need to recover their global position along the major axis by adding an offset equal to the total number of cell-slices in the previous slabs (line 7 of Algorithm 1).

In order to completely cover each valid ray segment, we then need to do some simple flipping and offsetting to calculate the voxel-slice indices ($startVoxel_{index}$ and $endVoxel_{index}$ shown as the red solid lines in Figure 5) from their corresponding cell-slice indices ($start_{index}$ and end_{index}) for each segment. Note that "cell-slice" refers to a slice of cube-shaped cells perpendicular to the major axis, while "voxel-slice" refers to a plane, parallel with this, that lies along the boundary faces of the cubes (as a cube has a voxel at each of its 8 corners).

If the camera is aligned with the major axis ($sign_{oa} = 1$), they are calculated as: $startVoxel_{index} = start_{index}$ and $endVoxel_{index} = end_{index} + 1$; otherwise ($sign_{oa} = -1$), we need to flip both indices in order to recover the correct position relative to $voxelSlice0$ (the first voxel-slice): $startVoxel_{index} = TotalSlice - start_{index}$ and $endVoxel_{index} = TotalSlice - 1 - end_{index}$ (lines 12-13 in Algorithm 1).

Having obtained the voxel-slice indices for a segment,

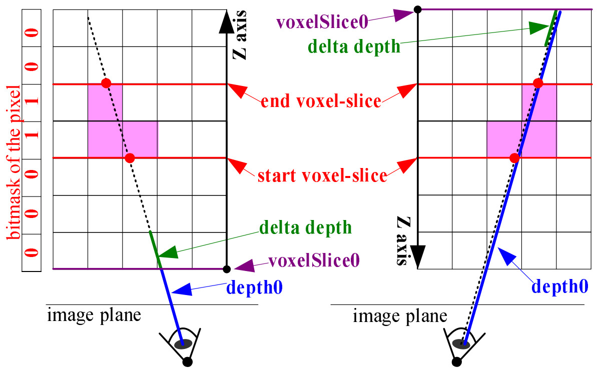

Figure 5: *The bitmask here is "0011000", so there is only one set bit-segment, whose start and end indices are originally 3 and 4, respectively. The left figure shows the case when the camera points along the major axis Z, so $startVoxel_{index} = 3$ and we need only to offset the end index by 1: $endVoxel_{index} = 4 + 1 = 5$; while the right figure shows the case when the camera points in the opposite direction to the major axis Z, in which case we need to flip both indices: $startVoxel_{index} = TotalSlice - 3 = 4$ and $endVoxel_{index} = TotalSlice - 1 - 4 = 2$, where $TotalSlice = 7$ in this case.*

we can compute their fp32 depth values (distances from camera along the ray) very easily. Firstly, for each pixel, we calculate $cosAlpha = DotProduct(\overrightarrow{rayDir}, \overrightarrow{axis_m})$, where $\overrightarrow{rayDir}$ is the ray direction from the camera. The distance from camera to $voxelSlice0$ along this ray is calculated as: $depth0 = (voxelSlice0 - camPos\_z)/cosAlpha$, where $camPos\_z$ is the component of $\overrightarrow{camPos}$ (the camera position vector) along the major axis. We also calculate a delta depth between any two neighbouring slices as $delta_{depth} = sideLength/cosAlpha$. In Figure 5, $depth0$ and $delta_{depth}$ are shown as the blue and green line segments, respectively. Having these two variables ready for each pixel, we can easily compute the depth values for the start and end voxel-slices using the following equations:

$$\begin{aligned} start_{depth} &= abs(depth0 + startVoxel_{index} * delta_{depth}) \\ end_{depth} &= abs(depth0 + endVoxel_{index} * delta_{depth}) \end{aligned} \quad (1)$$

4.3.2. Raycasting within valid ray segments

The start and end depths of a segment define a depth interval and border for a valid ray segment, and the two border points can be calculated as:

$$\begin{aligned} \overrightarrow{start_{pos}} &= \overrightarrow{camPos} + start_{depth} * \overrightarrow{rayDir} \\ \overrightarrow{end_{pos}} &= \overrightarrow{camPos} + end_{depth} * \overrightarrow{rayDir} \end{aligned} \quad (2)$$

Between the two border points of the ray segment, we can calculate the exact intersection of the ray with the isosurface by uniform stepping, which iteratively steps along the ray from $\overrightarrow{start_{pos}}$ to $\overrightarrow{end_{pos}}$ in a near-to-far order. This is performed until an isosurface is found or $\overrightarrow{end_{pos}}$ is reached. If an isosurface is found, we refine the position of the intersection by using a binary search [LCD09b]. If higher quality is desired, complex intersection methods such as the correct root finding method in [MKW*04] can be employed.

One important advantage of the algorithm is that, thanks to our bit flipping, all of the peeled depth values are naturally in a near-to-far order which obviates the need for explicit sorting. This is a huge saving, since sorting float-point depth values in a fragment shader is very time-consuming. Thus, whenever an isosurface intersection is found, we can be sure that it is the nearest one along the ray so we can safely exit the program. Furthermore, for each frame, $sign_{oa}$ is the same for all pixels, so no divergent branching takes place at runtime, and all pixels of a frame always take the same path without divergence. Hence it can fully utilize the SIMD efficiency of GPU. The maximum count of depth layers that our approach can peel is 16 times the layer number of the array texture. If we set the layer number of the array texture to 512, we can peel up to 8192 (512×16) depth layers, which is much larger than that of the MRT-based approaches, since current GPUs can support, at most, 8 MRTs.

5. Transparent Rendering of Multiple Isosurfaces

It is easy to extend our algorithm from an opaque first-hit isosurface to the transparent rendering of multiple isosurfaces, since all the valid ray segments generated are already in front-to-back order, which makes accurate transparent blending straightforward. After we find a ray-isosurface intersection, the program does not exit (line 21 of Algorithm 1), instead we blend the shading result to an accumulated colour - the final colour is then composited from several isosurfaces. Some results of transparent rendering are shown in Figure 6.

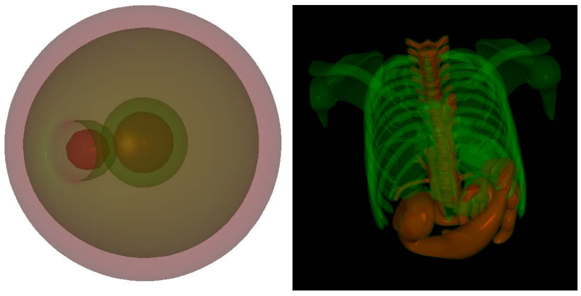
Figure 6: *Transparent rendering results. The left image shows the nucleon dataset (with resolution of 64^3) rendered at 71.9 fps; the right image shows the anatomia dataset (with resolution of 512^3) rendered at 21.3 fps.*

6. Results and Discussion

We tested our implementation using OpenGL/GLSL on a desktop PC (AMD 2.3 GHz CPU with 3 GB RAM) with an NVIDIA GeForce GTX285 graphics card with 1GB RAM. All the rendering and comparisons were performed on this computer at 1024×1024 screen resolution.

data set	size	isovalue	kdjump	ours	s
bonsai	256^3	40	16.6	73.9	4.5
aneurism256	256^3	48.5	19.8	95.6	4.8
aneurism512	512^3	50.5	19.2	82.7	4.3
foot	256^3	100	23.2	78.8	3.4
skull	256^3	48	13.3	52.2	3.9
xmasBox	512^3	35.5	14.9	54.2	3.6
xmasTree	512^3	35.5	17.2	59.1	3.4
backpack	512^3	30.5	12.8	50.4	3.9
stagbeelte	$1024^2 * 512$	39.5	n.p.	68.9	n.p.
fuel	64^3	2.5	6.66	241.4	36.2
hydrogenAtom	128^3	12.5	16.0	169.5	10.6

Table 1: *Performance in fps for static isovalues for kdjump and our algorithm; s is the speedup ratio of our method over kdjump.*

6.1. Quality and performance comparison

For performance evaluation, we compared our algorithm to recently published state-of-the-art isosurface-rendering algorithms. For the fairest comparison, we ran the various test cases under precisely the same viewing configurations for each comparison. All framerates were obtained by rendering a view multiple times to produce an average value.

6.1.1. Comparing with an image-order algorithm

kd jump is an optimised CUDA-based isosurface raytracing method [HL09], the source code for which was kindly provided by the authors. Without changing any algorithmic part of the code, we measured its performance using the same datasets, viewing configurations and hardware platform to ensure a fair comparison with our algorithm.

In the first test, the isovalue is fixed, so *kd jump* can skip the kd-tree rebuilding step and our algorithm can skip the HistoPyramid-building step; the resulting performance statistics are shown in Table 1, from which it is obvious that our method is much the faster.

In the second case, we assume that the isovalue changes for each frame, so that, at every frame, our algorithm must rebuild the HistoPyramid and *kd jump* has to rebuild the kd-tree. From Table 2, we can see that, for large datasets, our performance advantage over *kd jump* is more evident for dynamic isovalues due to the higher cost associated with kd-tree rebuilding. Dynamic isovalues arise when the users need to change the isovalues frequently at runtime in order to explore the different structures in the dataset. The rendered images for these two tables are shown in Figures 7 and 8.

The quality comparison can be seen in Figure 7. Both algorithms use a similar uniform stepping to find the isosurface intersection (using the same stepping distance, that is, half of the voxel size) and central differences for on-the-fly normal calculation, but our algorithm also uses a binary

data set	size	isovalue	kdjump	ours	*s*
bonsai	256^3	40	12.1	68.9	5.7
aneurism256	256^3	48.5	13.7	84.6	6.2
aneurism512	512^3	50.5	4.57	72.8	15.9
foot	256^3	100	15.2	71.9	4.7
skull	256^3	48	10.4	49.5	4.8
xmasBox	512^3	35.5	4.3	50.4	11.7
xmasTree	512^3	35.5	4.4	55.2	12.5
backpack	512^3	30.5	4.1	48.2	11.8
stagbeelte	$1024^2 * 512$	39.5	n.p.	63.0	n.p.
fuel	64^3	2.5	6.59	209.9	31.9
hydrogenAtom	128^3	12.5	15.4	153.7	9.98

Table 2: *Performance in fps for dynamic isovalues for kdjump and our algorithm, so all the isovalue-related variables need to be recomputed from scratch for each frame; s is the speedup ratio of our method over kdjump.*

data set	size	isovalue	*cuda_mc*	ours	*s*
bonsai	256^3	40	38.7	68.9	1.8
aneurism256	256^3	48.5	56.3	84.6	1.5
aneurism512	512^3	50.5	n.p.	72.8	n.p.
foot	256^3	100	41.2	71.9	1.7
skull	256^3	48	31.5	49.5	1.6
xmasBox	512^3	35.5	n.p.	50.4	n.p.
xmasTree	512^3	35.5	n.p.	55.2	n.p.
backpack	512^3	30.5	n.p.	48.2	n.p.
stagbeelte	$1024^2 * 512$	39.5	n.p.	63.0	n.p.
fuel	64^3	2.5	256.3	209.9	0.8
hydrogenAtom	128^3	12.5	168	153.7	0.91

Table 3: *Performance in fps for dynamic isovalues for both cuda_mc and our algorithm; s is the speedup ratio of our method over cuda_mc. For large datasets with resolution of* 512^3 *or larger, the cuda_mc program always crashes due to running out of CUDA memory, which is referred to as "n.p.".*

search to refine the intersection. Furthermore, $kdjump$ employs only simple diffuse lighting, whereas our method uses full Phong shading (including diffuse and specular lighting). Our rendering results look smoother than those of $kdjump$ due to these additional computations in our raycaster. Even though our raycaster is more complex than that of $kdjump$, our performance still is much faster, as mentioned above. For very large datasets, such as $stagbeetle$, the $kdjump$ program always crashes as a result of running out of CUDA memory, which is referred to as "n.p." in Tables 1 and 2, which demonstrates that ours has a smaller memory footprint.

6.1.2. Comparing with another image-order algorithm

Knoll et al. [KHW*09] recently introduced a method for rendering isosurfaces as Dirac impulses by using peak finding. They employed a 3D DDA algorithm for empty-space skipping and adaptive sampling, followed by a binary search to find the isosurfaces. We do not have their source code, so it is not possible to compare with them directly, but fortunately their paper also tested the two public datasets *aneurism*256 and *backpack*, and their performance was measured on the same GPU (GTX285), so we can make an indirect comparison. Their reported framerates for the two datasets are 5.3fps and 2.1fps, respectively, while ours are 95.6fps and 50.4fps, respectively, which are around 20 times faster. So it seems that our empty-space skipping is much more efficient than their image-order approach.

6.1.3. Comparing with an object-order algorithm

The object-order algorithm against which we test our method is a CUDA-based isosurface rendering using Marching Cubes [NVI09]), for which the source code is publicly available. We refer to it as *cuda_mc*. It first runs an efficient scan function (prefix sum) from the highly optimised CUDPP library to perform stream compaction, that is, extraction of all of the active cells, and then uses MC to generate triangles. Its rendering quality is clearly worse than ours (Figure 7), since it produces a faceted appearance (diamond artefacts), caused by its planar triangles, while our approach uses ray-isosurface intersection to produce a smoother result. The resulting performance statistics are shown in Table 3. Our performance is faster than *cuda_mc* except for the last two datasets, which have a very small data size.

6.2. Discussion

Our algorithm is a hybrid [RYL*96]: the rasterisation pass is in object order and the raycasting pass is in image order - so it inherits advantages from both approaches. For image-order methods, performance is in almost direct proportion to the screen resolution, whereas in object-order methods, performance is relatively independent of it. Thus, for a certain scene complexity, object-order methods suffer little impact if the screen resolution is increased. In the $fuel$ dataset, for which our method is more than 30 times faster than $kdjump$ (Tables 1 and 2), there is little geometry to rasterise owing to the low resolution of this dataset. In contrast, $kdjump$ always traces each individual pixel as usual. As a result, when the screen resolution is high, it will be very slow, even though the dataset itself is simple. With regard to the slow framerate for this dataset, we received a helpful communication from the authors of [HL09], who thought this was due to the dataset itself since some datasets can create special difficulties for kd-tree and may cause it to traverse for a long time.

As a fully object-order method, *cuda_mc* is faster than our algorithm for very small-sized datasets (Table 3), but our approach has the advantages of an image-order method so its quality is obviously higher than that of *cuda_mc*; further, for larger datasets, our algorithm has not only better rendering quality, but also faster speed. Our method has a particular advantage over *cuda_mc* when the isosurfaces have many depth layers that all contribute to the final transparent rendering before the rays reach saturation, as shown in Figure 6. Such a case requires that all of the isosurface layers must be rendered in a view-dependent order [CICS05], so the triangles output from Marching Cubes need to be sorted, which is

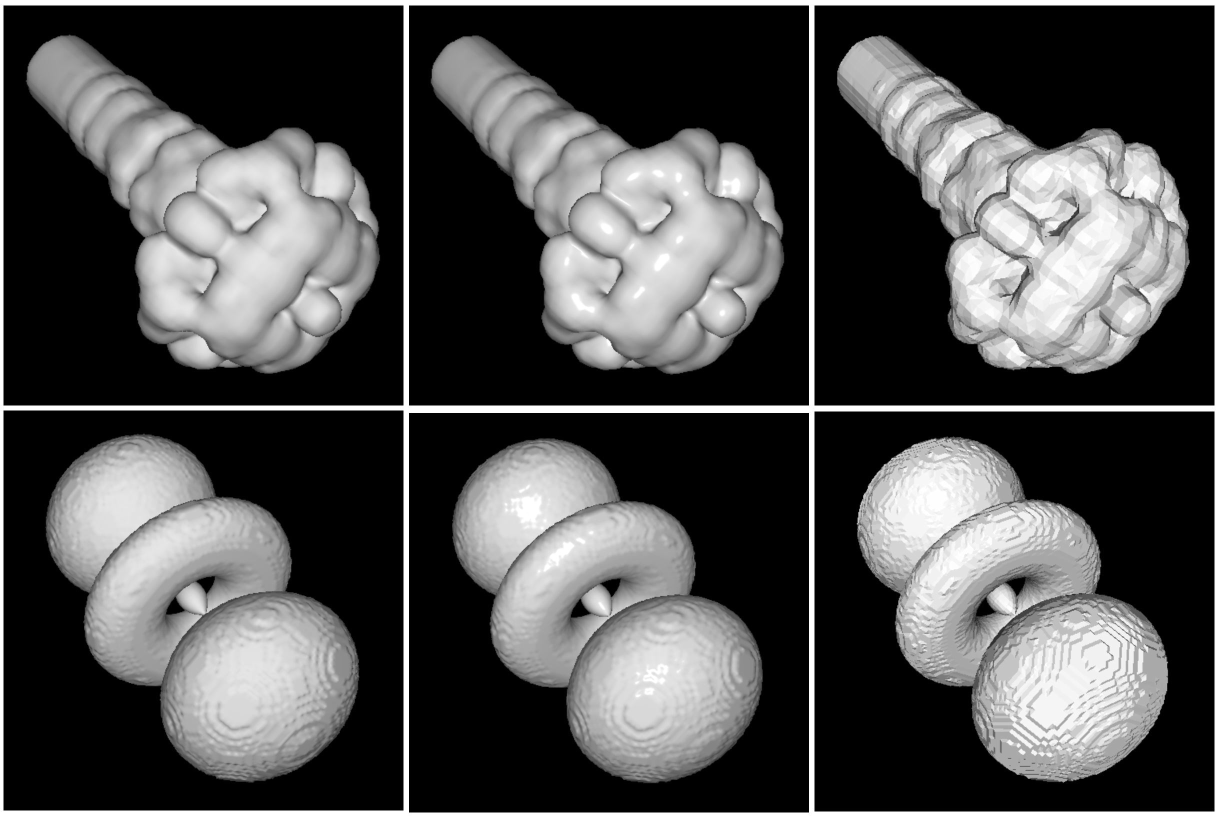

Figure 7: *Quality comparison between kdjump, our algorithm and cuda_mc, using the fuel and hydrogenAtom datasets. The rendering quality of cuda_mc is worse than ours due to its faceted appearance (diamond artefacts); kdjump uses only uniform stepping to find the intersection; while our algorithm uses a similar uniform stepping, but this is followed by a binary search to refine the intersection. Hence our results are slightly smoother than those of kdjump. Furthermore, kdjump uses only simple diffuse lighting, while ours uses full Phong shading.*

very demanding and may dominate the total rendering time if the number of triangles is large [GHLM05]. In contrast, our approach generates all of the valid ray segments immediately in a near-to-far order, which makes transparent rendering of multiple layers of isosurfaces simple and straightforward.

To fully explore the performance of the technique, we timed the 3 individual stages of the algorithm and found that the potential bottleneck lies in the third stage, which accounts for 85% of the total rendering time for one frame in a typical dataset, while the first and second stages account for 7% and 8%, respectively.

To better evaluate and demonstrate the importance of the empty-space skipping of interior regions, we modified the algorithm to preserve only the first and last depth layers, as in previous methods, while keeping all other parts of the algorithm the same. In doing so, only exterior empty space is skipped, and we found that the framerate dropped by 36% for typical datasets. So, it appears that the ability to skip interior regions plays an important role in GPU raycasting.

6.3. Cell granularity

Our basic cells are the MC cells (each with 8 scalar-value corners), but we can increase the cell granularity *granu_c* in order to reduce the cell count and generate more efficient computation. Setting *granu_c* to 1 gives the basic MC cells; otherwise, we have macro-cells, each composed of $granu\_c^3$ basic MC cells. We found that different datasets may need different cell granularity for optimal performance. In our tests, *granu_c* for the datasets: *bonsai*, *aneurism*256, *aneurism*512, $foot$, $skull$, *xmasBox*, $xmasTree$, *backpack*, $stagbeetle$, $fuel$, and *hydrogenAtom* are 2, 2, 4, 2, 2, 4, 4, 4, 8, 1 and 2, respectively.

7. Conclusion

To accelerate GPU isosurface raytracing, we have created a multi-layer depth-peeling approach that uses only a sin-

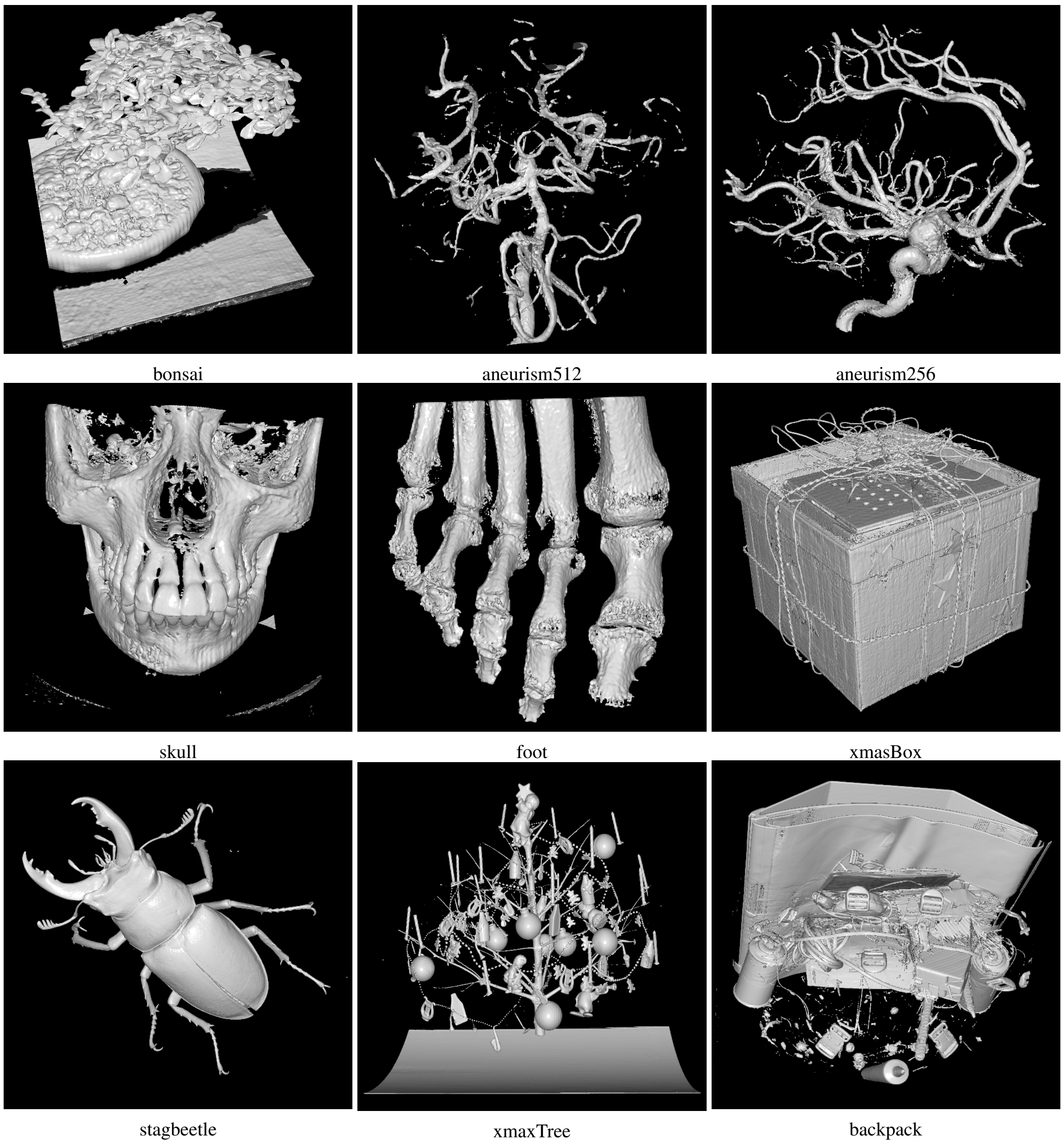

Figure 8: *Rendering results of our algorithm for different dense and sparse data sets. For these viewing configurations, the corresponding performance statistics of our algorithm, kdjump and cuda_mc are shown in Tables 1, 2 and 3.*

gle hardware rasterising pass. In this, we immediately obtain the depth layers of the tightly-fitting bounding geometry of the isosurface, the maximum number of which can be in the thousands, with each layer in fp32 accuracy. By raytracing only the valid ray segments that lie between pairs of consecutive depth layers, we can skip both the interior and exterior empty space efficiently.

Acknowledgments

We would like to thank David M. Hughes and Ik Soo Lim for the valuable discussion and for making their *kd jump* code available, which was instrumental in enabling us to perform comparisons under the same conditions. We thank also the anonymous reviewers for their constructive comments and suggestions. This work was partially supported by the European Commission within the Marie Curie project GAM-VolVis (FP7-PEOPLE-IIF-2008-236120).

References

[BCL*07] BAVOIL L., CALLAHAN S. P., LEFOHN A., COMBA JO A. L. D., SILVA C. T.: Multi-fragment effects on the GPU using the k-buffer. In *I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), ACM, pp. 97–104.

[BM08] BAVOIL L., MYERS K.: Order independent transparency with dual depth peeling. NVIDIA OpenGL SDK, 2008.

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L. D., SILVA C. T.: Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–295.

[ED06] EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *I3D '06: Proceedings of the 2006 Symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), ACM, pp. 71–78.

[ED08] EISEMANN E., DÉCORET X.: Single-pass GPU solid voxelization for real-time applications. In *GI '08: Proceedings of Graphics Interface 2008* (Toronto, Ont., Canada, Canada, 2008), Canadian Information Processing Society, pp. 73–80.

[Eve01] EVERITT C.: *Interactive Order-Independent Transparency*. Research report, NVIDIA Corporation, 2001.

[GHLM05] GOVINDARAJU N. K., HENSON M., LIN M. C., MANOCHA D.: Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *I3D '05: Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2005), ACM, pp. 49–56.

[HL09] HUGHES D. M., LIM I. S.: Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1555–1562.

[HLSR09] HADWIGER M., LJUNG P., SALAMA C. R., ROPINSKI T.: Advanced illumination techniques for GPU volume raycasting. In *SIGGRAPH '09: ACM SIGGRAPH 2009 courses* (New York, NY, USA, 2009), ACM, pp. 1–166.

[HQK05] HONG W., QIU F., KAUFMAN A.: GPU-based object-order ray-casting for large datasets. In *Volume Graphics 2005* (2005), pp. 177–185.

[KHW*09] KNOLL A., HIJAZI Y., WESTERTEIGER R., SCHOTT M., HANSEN C., HAGEN H.: Volume ray casting with peak finding and differential sampling. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1571–1578.

[KW03] KRÜGER J., WESTERMANN R.: Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003* (2003), pp. 287–292.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 163–169.

[LCD09a] LIU B., CLAPWORTHY G. J., DONG F.: Accelerating volume raycasting using proxy spheres. *Computer Graphics Forum 28*, 3 (2009), 839–846.

[LCD09b] LIU B., CLAPWORTHY G. J., DONG F.: Fast isosurface rendering on a GPU by cell rasterisation. *Computer Graphics Forum 28*, 8 (2009), 2151–2164.

[LHLW09a] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Efficient depth peeling via bucket sort. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 51–57.

[LHLW09b] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Single pass depth peeling via CUDA rasterizer. In *SIGGRAPH '09: SIGGRAPH 2009: Talks* (New York, NY, USA, 2009), ACM, pp. 1–1.

[LWX06] LIU B., WEI L.-Y., XU Y.-Q.: *Multi-layer depth peeling via fragment sort*. Research report, msr-tr-2006-81, Microsoft Research Asia, 2006.

[MB07] MYERS K., BAVOIL L.: Stencil routed A-buffer. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches* (New York, NY, USA, 2007), ACM, p. 21.

[MKW*04] MARMITT G., KLEER A., WALD I., FRIEDRICH H., SLUSALLEK P.: Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *VMV04* (2004), pp. 429–435.

[NVI09] NVIDIA: Nvidia CUDA SDK 2.3 code samples, 2009.

[RBE08] ROESSLER F., BOTCHEN R. P., ERTL T.: Dynamic shader generation for GPU-based multi-volume ray casting. *IEEE Comput. Graph. Appl. 28*, 5 (2008), 66–77.

[RYL*96] REED D. M., YAGEL R., LAW A., SHIN P.-W., SHAREEF N.: Hardware assisted volume rendering of unstructured grids by incremental slicing. In *VVS '96: Proceedings of the 1996 Symposium on Volume Visualization* (Piscataway, NJ, USA, 1996), IEEE Press, pp. 55–62.

[SA95] SOBIERAJSKI L. M., AVILA R. S.: A hardware acceleration method for volumetric ray tracing. In *VIS '95: Proceedings of the 6th Conference on Visualization '95* (Washington, DC, USA, 1995), IEEE Computer Society, pp. 27–34.

[VMD08] VIDAL V., MEI X., DECAUDIN P.: Simple empty-space removal for interactive volume rendering. *J. Graphics Tools 13*, 2 (2008), 21–36.

[ZTTS06] ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.-P.: On-the-fly point clouds through histogram pyramids. In *VMV06* (2006), pp. 137–144.