Parallel Peeling of Curvilinear Grids

Sören Grimm*
Vienna University of Technology
Austria

Michael Meissner†
Viatronix Inc.
NY, USA

Armin Kanitsar‡
Tiani Medgraph AG
Vienna, Austria

Eduard Gröller§
Vienna University of Technology
Austria

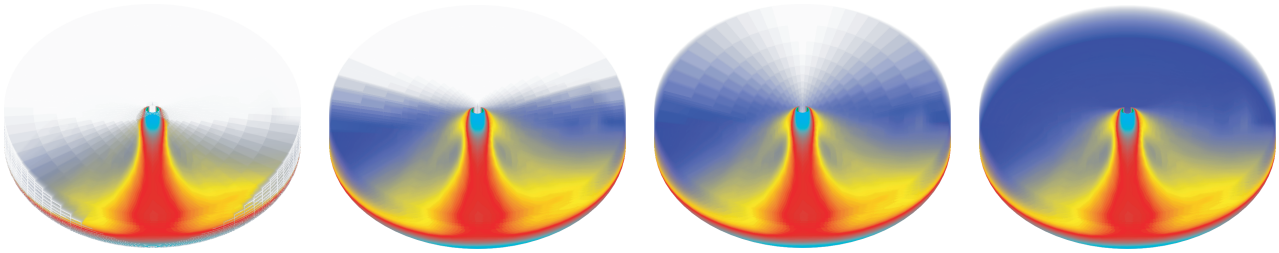

Figure 1: Liquid Oxygen Post: Four iteration steps of our parallel peeling algorithm.

ABSTRACT

In this paper we present a novel hybrid CPU-GPU approach for rendering curvilinear grids. Visibility sorting is accomplished by parallel peeling cells off the grid, utilizing an active cell peeling front. In each step, we compute the ray-cell intersection coordinates on the GPU, perform accurate volume integration (CPU), and determine the set of active cells for the next iteration (GPU). The approach requires only standard graphics capabilities and can therefore be used on any commodity PC, including laptops. Furthermore, the main memory requirements are negligible since the required data structures are minimal.

The main advantage of our algorithm is that we exploit hardware acceleration for the expensive visibility sorting which is beneficial over time due to the faster performance increase of GPUs over CPUs. Due to the simplicity of the algorithm and its low requirements on preprocessing and main memory, it is well suited for thin clients. Last but not least, the approach could easily be extended to irregular grids using tetrahedra.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: Curvilinear grids, Raycasting, Volume Rendering, Visibility Sorting, Depth Sorting

1 INTRODUCTION

The curvilinear grid is one of the most common volume data formats in application areas such as scientific computing and computer-based modelling. For example, it is used in computational fluid dynamics or partial differential equation solvers. This type of grids can be seen as the result of a non-linear transformation applied on a rectilinear grid. Hereby the topological structure is preserved. However, the implicit storing of grid positions is lost due to the arbitrary alignment of grid points. Each 3D location of a grid point has to be stored explicitly. The non-linear transformation implies in general non-uniformly shaped cells. This introduces considerable complexity to the rendering of such volumetric grids. In general, the evaluation of volume rendering equations requires visibility sorting of cells intersected by rays [11]. For each image pixel a ray is cast through the volume and needs to be integrated in the correct visibility order. Since the visibility order on a curvilinear grid is not implicitly given, it has to be computed explicitly. In this paper we present an alternative method to obtain this visibility ordering. We apply a peeling approach that utilizes standard commodity graphics hardware.

*e-mail: grimm@cg.tuwien.ac.at
†e-mail: meissner@viatronix.com
‡e-mail: kanitsar@tiani.com
§e-mail: groeller@cg.tuwien.ac.at

2 RELATED WORK

Rendering of curvilinear and unstructured grids is quite a challenging task, due to their topology. One well known acceleration technique is the Projected Tetrahedra (PT) algorithm [13]. In general one of the main challenges in rendering such grids is the inherent need to sort the cells according to visibility. Stein et al. [17] proposed an exact visibility ordering algorithm with complexity $O(n^2)$. De Berg et al. [3] proposed a faster approach which runs with $O(n^{4/3+\varepsilon})$ complexity. Other approaches to perform the necessary visibility sorting have been proposed by Williams et al. [21] Meshed Polyhedra Visibility Ordering (MPVO) or Cook et al. [2]. Silva et al. [16] (XMPVO) extends the MPVO algorithm such that it efficiently generates an exact visibility ordering for arbitrary polyhedra cell complexes in interactive time. There are also several approaches which reduce the 3D sorting problem to a 2D problem within each plane Silva et al. [14], Giertsen et al. [5], Silva et al. [15], Yagel et al. [23] and Wilhelms at al. [20]. Furthermore Bunyk et al. [1] presented a simple and efficient ray casting engine for grids composed of tetrahedra cells, or other cell complexes where cells have been broken up into faces. The visibility determination in their approach is done in screen space. For each pixel an ordered list of boundary faces is computed. The final rendering is done performing ray casting. Hong et al. [7] presented an efficient robust ray-casting algorithm for directly rendering a curvilinear volume of arbitrary shaped cells. He improved his approach in [8]. Farias et al. [4] (ZSWEEP) propose an approach based on sweeping the data with a plane parallel to the viewing direction. There are also several approaches exploiting the graphics hardware for volume rendering of unstructured grids. Westermann et al. [19] proposed a sweep-

plane approach which is accelerated by hardware assisted polygon drawing. Guthe et al. [6] proposed an approach for accurate rendering of unstructured grids using multi-texturing. High quality colors and opacities of the pre-integration table are achieved by exploiting the high internal precision of the pixel shader. Roettger et al. [12] proposed a hardware-accelerated pre-integration approach and a rendering method which utilizes 2D texture mapping instead of 3D texture mapping. Weiler et al. [18] proposed an approach which performs all computations for the projection and scan conversion of a set of tetrahedra on the graphics hardware.

Our novel approach utilizes both GPU and CPU and in contrast to the previous approaches, sorting is implicitly performed on the GPU using an iterative parallel peeling algorithm. Due to its simplicity, our approach does not require costly pre-processing nor do we require large data structures to be kept in main memory. The interpolation of the data and depth values is performed on the GPU and only the actual volume integration is performed on the CPU.
The presentation of our approach is organized as follows: In Section 3 we present our peeling approach. We subdivided the different steps of our method in pre-processing, entry-cell determination, peeling and compositing. At the end of the section we show a peeling work-flow example. The results are presented in Section 4 and finally in Section 5 we conclude our work.

3 Parallel Grid Peeling

Within this section, we will describe the details of our parallel grid peeling algorithm. It consists of a one-time only pre-processing step for loading the grid and decomposing its cell faces into triangles. For each frame, we need to determine the ray entry point(s) on a per ray basis and advance the rays through the grid from cell to cell in what we call iterative parallel peeling.

3.1 Pre-processing

The pre-processing step is performed only once for each curvilinear grid that is to be displayed. Since we are using standard capabilities of the GPU, the curvilinear grid needs to be decomposed into triangles that can be rendered. A cell generally consists of six faces and each of the faces is decomposed into two triangles, resulting in a total of twelve triangles per cell. For each triangle, we assign a unique ID and by keeping these IDs consecutive and in a certain order[1], the ID of the neighboring cell as well as the ID of the neighboring triangle are implicitly given:

$$
\begin{aligned}
Cell\_ID &= Triangle\_ID \,/\, 12 \\
Triangle\_ID_{relative} &= Triangle\_ID - 12 * Cell\_ID \\
Cell\_ID_{neighbour} &= Cell\_ID + LUT(Triangle\_ID_{relative}) \\
Triangle\_ID_{neighbour} &= 12 * Cell\_ID_{neighbour} \\
&\quad + ((Triangle\_ID_{relative} + 6) \,\%\, 12)
\end{aligned}
$$

The LUT to obtain the cell offset is:

$$
\begin{array}{lcl}
0,1 & \Rightarrow & \text{-1} \\
2,3 & \Rightarrow & \text{-}\#Grid\_Cells_X \\
4,5 & \Rightarrow & \text{-}\#Grid\_Cells_X * \#Grid\_Cells_Y \\
6,7 & \Rightarrow & 1 \\
8,9 & \Rightarrow & \#Grid\_Cells_X \\
10,11 & \Rightarrow & \#Grid\_Cells_X * \#Grid\_Cells_Y
\end{array}
$$

[1]Triangles are indexed 0 through 11 starting with the cell face pointing to the negative X-axis, negative Y-axis, negative Z-axis, positive X-axis, positive Y-axis, and positive Z-axis. All axes with respect to the index orientation of the original grid.

with the restriction that the $Cell\_ID_{neighbour}$ can only be computed as described above if the current cell ($Cell\_ID$) is not an outer cell. Outer cells are defined as cells which contain at least one face which does not have a neighboring cell. Vice versa, inner cells are defined as cells which have adjacent neighbor cells for each cell face. Determining whether a cell belongs to an outer cell can easily be done by comparing its axis indices x, y, and z index with 0 and the maximum number of cells along each axis.
In addition to the triangle decomposition, we also compute a normal for each triangle. It is computed such that it always points to the outside of the cell, as indicated in Figure 2). While we could compute the normals on the fly using cross products, we are precomputing them and trading memory for performance. The normals are needed in order to increase performance while finding entry points for the given cells through backface culling (see Figure 2(b)) and to find the exit points using frontface culling (see Figure 2(c)).

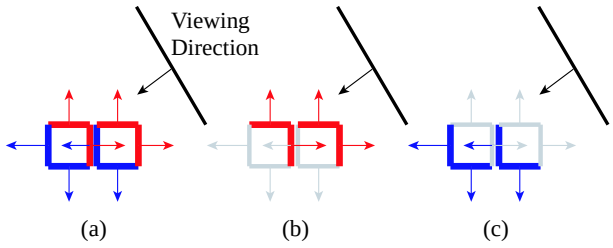

Figure 2: Culling: (a) Each cell is defined such that the normal vectors point outwards a cell. (b) Visible faces (red) for a given viewing direction, backface culling enabled. (c) Visible faces (blue) for a given viewing direction, frontface culling enabled.

3.2 Entry and Re-Entry Points

For every frame to be generated, the ray entry points need to be determined. Depending on the topology of the curvilinear grid and the actual view point, each ray could have multiple entry points into the curvilinear grid by re-entering again. For common curvilinear grids such as in the blunt fin, post, or delta wing datasets, there is up to one potential re-entry per ray, as shown in Figure 3. For these type

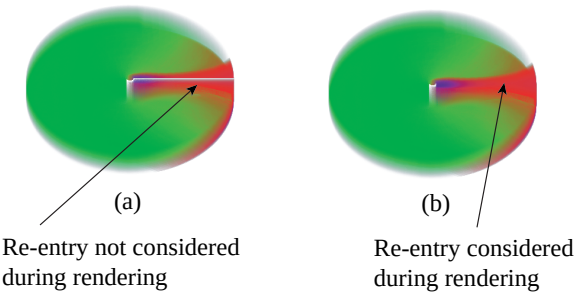

Figure 3: Liquid Oxygen Post Data. (a) Re-entry not considered during rendering. (b) Re-entry considered during rendering.

of grids, we can use a GPU accelerated implementation which simply renders all triangles that belong to the outer cells of the curvi-

linear grid[2]. To find the entry points, we enable backface culling and render each triangle. By setting the depth test to *GL_LESS*, we obtain for each screen pixel (ray) the closest entry point into the grid Vice versa, setting the depth test to *GL_GREATER* yields the potential re-entry point. While this approach works well for up to two entry points per ray (one re-entry), we currently require a software implementation to compute the entry points of curvilinear grids with more than one re-entry per ray. During the actual rendering of the outer cell triangles, we are shading each triangle with its unique *Triangle_ID*, see Figure 4(b). The resulting frame-buffer

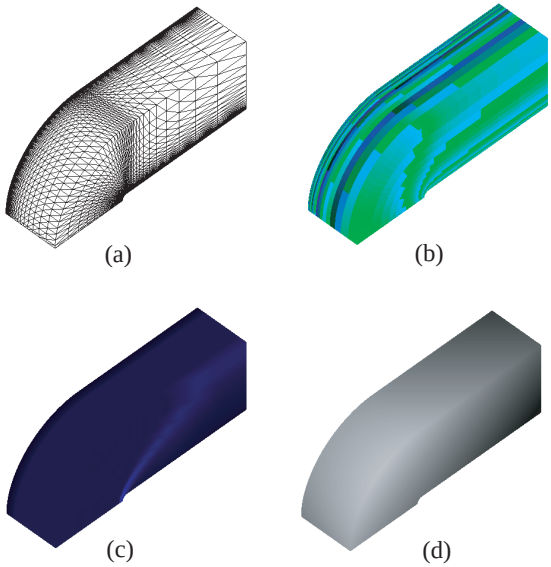

Figure 4: Blunt Fin dataset: (a) Triangles rendered in line mode. (b) Triangles shaded with cube IDs. (c) Triangles shaded with data values. (d) Depth buffer image of triangles (brighter is closer).

image contains for each screen pixel the ID of the triangle and the depth value. These values are read and used to set up the initial active triangle and cell that each pixel is initially holding on to. A re-entry for a pixel is found if flipping the depth test yields a different *Triangle_ID*. The depth information is used as initial depth for the rays and by evaluating the occurring triangle IDs the entry cells are determined. For each of the two runs we introduce an additional rendering step in order to render the triangles using the actual data values specified at the grid locations. Those are also read and used to initialize the data for each entry point. The described steps correspond to Figure 6, row 1-9. Finally, we run-length encode (RLE) the image space Levoy[9] to be able to quickly skip processing of empty pixels during the actual peeling process. The RLE is updated for every pixel that is completed during peeling so that we can make full use of completed rays. Rays may complete when exiting the volume without any further re-entry or due to early ray termination.

3.3 Parallel Peeling

We perform image synthesis by iteratively peeling off grid cells from the curvilinear grid which simultaneously advances all rays through the grid. For each cell that a ray has entered, it has to exit this cell through one of the other faces before it can enter the next adjacent cell. The peeling progress for a rectilinear grid is illustrated in Figure 5. Figure 5(a) shows the given grid and the viewing

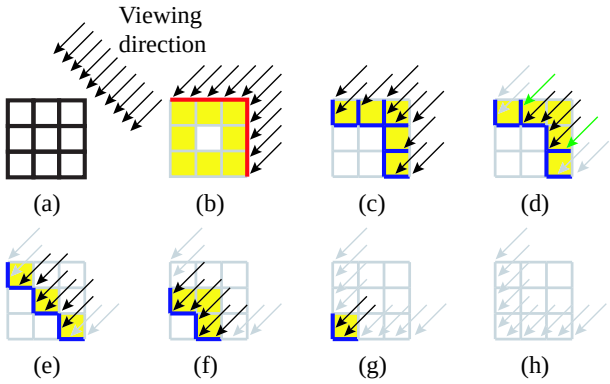

Figure 5: Peeling progress illustrated on rectilinear grid: Active cells are shown in yellow. (a) Given rectilinear grid and rays. (b) Determine entry of rays. Red: outer hull of grid. Backface culling is enabled. (c) - (h) Peeling of grid and processing of rays. Blue: current active faces/cells, black: active ray which will advance in next iteration, green: ray that temporarily cannot advance due to other rays that are "behind", and inactive cells and finished rays are shown in grey. Frontface culling is enabled.

direction. Figure 5(b) illustrates the previous determination of the entry cells. In each subsequent iteration a list of current active cells is generated. A cell is considered active if at least one ray holds on to it, waiting for an exit to be found. In Figure 5(c) the current active cells right after the initial entry determination are shown in yellow and their visible faces are shown in blue. The cells are rendered with frontface culling enabled (Figure 2(c)) and the resulting image as well as its depth footprint are read from the graphics card. Each ray determines if it hit the expected cell and in case it did, it will no longer hold on to this cell but to the adjacent one. Furthermore, the current depth value of the rays is used to perform the volumetric integration. Should the pixel reach the early ray termination criteria, the RLE of the image is updated. In case the ray hit an outer-cell the ray terminates as illustrated in Figure 5(d) as indicated by the greyed-out rays. In case a re-entry was pre-determined for that ray (see Section 3.2 it would be setup to the corresponding cell. Last but not least, in case a non expected triangle index is obtained, the ray will not advance in this iteration. This scenario is illustrated in Figure 5(d) by the green rays and due to neighboring rays being behind, needing to catch. Once all rays (pixels) have been processed, one peeling iteration is complete. Row 12 - 18 in Figure 6 summarize the overall algorithm and the update list of active cells is used for the next iteration until the list of active cells is empty which is equivalent to the RLE encoding jumping over all pixels. Due to the nature of this algorithm, all rays are advancing through the grid in parallel until all grid cells have been peeled off which lead to its name ”parallel peeling”.

While Figure 5 describes the iterative parallel peeling based on a rectilinear grid, it works exactly the same for a curvilinear grid (see Figure 7). However, there are some restrictions for certain curvilinear grids that contain non-convex cells. Non-convex cells are special cases since a given ray can possibly enter a cell more than once and as for most other algorithms, e.g. Max et al. [10], we need to address this scenario. Figure 8 illustrates a row of non-convex cells and one sample ray traversing one of them. First, the ray enters the middle cell, exits it on the back to enter into the cell that would be behind. Once it exits from this cell, it then re-enters the current cell. The issue here is that since we are rendering all front-

[2]For viewpoints inside the grid, we need to render all cells in order to determine the first entry point.

```
    // Entry Cell Determination
01: Enable Back-Face Culling
02: Create initial render-list: all outer-hull cells
03: Render render-list: shaded with cell IDs
04: Retrieve cell IDs
05: Retrieve depth information
06: Render render-list: shaded with data values
06: Retrieve shaded data values
07: Create new render-list: all cells entered by a ray
    // Peeling and volume integration
08: Enable Front-Face Culling
09: Repeat 3-7 once to determine ray re-entry
10: do
11: {
12:     Render render-list: shaded with cell IDs
13:     Retrieve cell IDs
14:     Retrieve depth information
15:     Render render-list: shaded with data values
16:     Retrieve shaded data values
17:     Perform volume integration
18:     Create new render-list: all cells which need to be
        exited by a ray in the next iteration
19: } while (render-list is not empty)
```

Figure 6: Pseudo-code of our peeling algorithm.

faces of a cell to determine the entry and all backfaces to determine the exit, the ray will get stuck since it will always find the entry point where it entered the cell the first time. In order to solve this problem, we split non-convex cells into tetrahedra so that we again end up with convex cells. While it creates some overhead in finding and activating neighboring cells, we simply mark non-convex cells and handle those in an "else-branch". Implicit neighborhood information as presented earlier on is still valid on a cell basis and within non-convex cells we again have an implicit ordering of the tetrahedras.

3.4 Volume Integration

Since we are not only interested in getting the ID of the involved triangle and cell but also the actual interpolated data value across the cell faces, we render each triangle twice: once with the ID and once with the actual data values at the vertices. Thus, the bilinear interpolation of the data values is done on the graphics card. To get the interpolated data the image is read from the graphics card, as shown in Figure 4(d). Once we do have the data and depth value of each ray in their current cell, we can perform volume integration on the CPU. In order to improve image quality, we do not simply composite a constant color across the cell length but perform sampling within each cell using a data set specific global sampling rate that accounts for the minimum and maximum cell sizes. The interpolated values are then accumulated:

$$\begin{aligned} I_R &= I_R + \alpha \cdot C_\alpha[val] \cdot C_R[val] \\ I_G &= I_G + \alpha \cdot C_\alpha[val] \cdot C_G[val] \\ I_B &= I_B + \alpha \cdot C_\alpha[val] \cdot C_B[val] \\ \alpha &= \alpha - \alpha \cdot C_\alpha[val] \end{aligned}$$

Herby $I_{R,G,B}$ is the resulting accumulated color, α the accumulated transparency, C_α the current opacity, $C_{\{R,G,B\}}$ the current RGB color values, and *val* the corresponding interpolated data-value. Instead of sampling along too long intervals and accumulating these into the final pixel value, we could alernatively apply some sort of pre-integration technique, e.g. as presented in [6, 12].

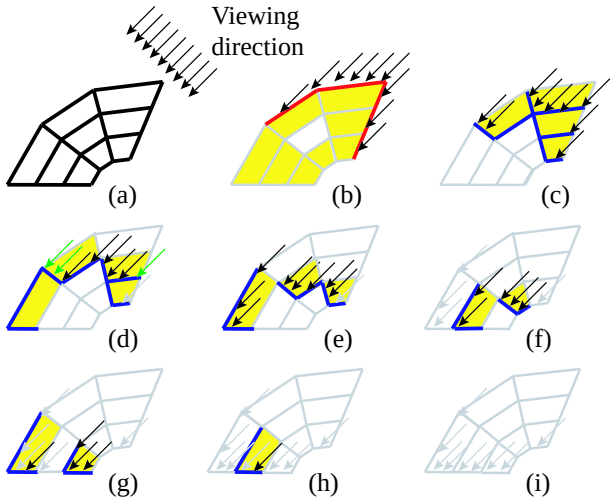

Figure 7: Peeling progress illustrated on a curvilinear grid: Active cells are shown in yellow. (a) Given curvilinear grid and rays. (b) Determine entry of rays. Red: outer hull of grid Backface culling is enabled. (c) - (i) Peeling of grid and processing of rays. Blue: current active faces/cells, black: active ray which will advance in next iteration, green: ray that temporarily cannot advance due to other rays that are "behind", and inactive cells and finished rays are shown in grey. Frontface culling is enabled.

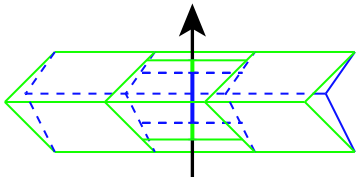

Figure 8: Ray may enter a non-convex cell more than once.

3.5 Peeling Illustration

For a better illustration of the iterative parallel peeling approach, four iterations steps during rendering of the Blunt Fin data set are shown in Figure 9. On the left side the current rendered cells are shown and on the right side the so far accumulated image. Figure 9(a) is the first iteration where compositing is applied. In Figure 9(b) a portion of the grid is already peeled off. The cavity on the left side corresponds to the part on the right side where the progression of the composting is quite advanced. Also early ray termination can be applied as shown in Figure 9(c). The hole in the grid corresponds to rays which already finished due to full accumulated opacity. These rays do not contribute any cells to the current render-list of cells. Finally in Figure 9(d) the result image can be seen.

4 Results

Several data sets were selected to validate the correctness of our approach as well as to measure its performance. The following table lists these data sets, their grid type, and the number of cells they consist of: The system we used to measure the performance is a laptop with Pentium Centrino CPU and GeForce4 4200 Go (32 MB). While the system is equipped with 1 GB of main memory, we

View (Figure 10)	Time [secs]	GPU [secs]	GPU [%]	CPU [secs]	CPU [%]	# Rendered cells	# Iterations	Sample Distance	Memory [MB]
(a)	1.10	0.62	56.4	0.48	43.6	48,275	75	0.50	2.34 MB
(b)	1.41	0.84	59,6	0.57	40.4	86,157	115	0.01	6.37 MB
(c)	1.47	1.35	91,8	0.12	8.2	140,778	204	0.10	12.44 MB
(d)	8.31	7.20	88,0	1.11	12.0	1,016,262	184	0.10	15.45 MB

Table 1: Statistics of the iterative peeling algorithm: Image size 256x256, rendered on a Pentium Centrino with GeForce4 4200 Go (32 MB)

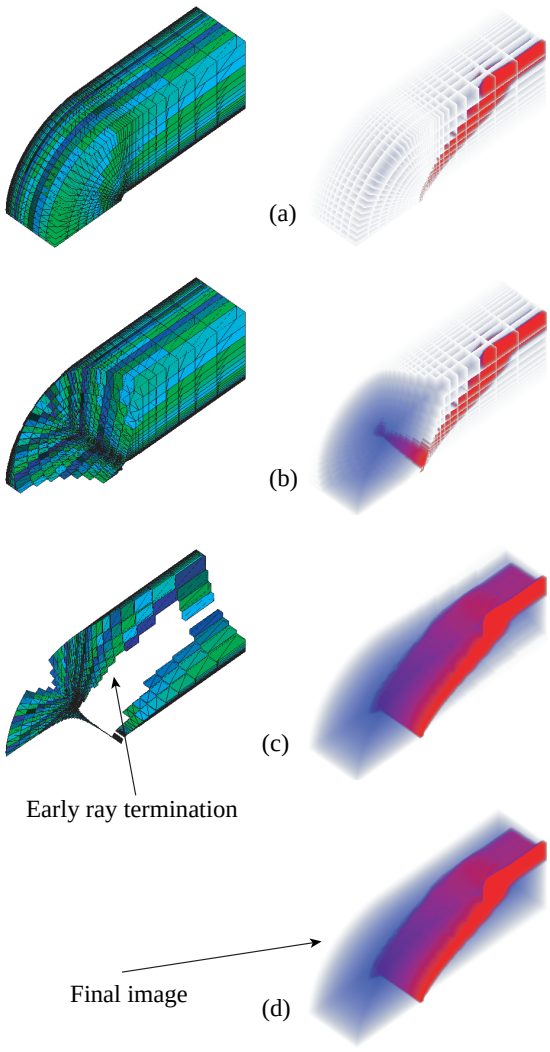

Figure 9: Iterative peeling of the blunt data set. Left column shows remaining cells and right column shows so far accumulated image. (a) through (d) illustrate different time steps of the 80 iterations.

never used more than 16 MB of main memory to run the algorithm.

4.1 Performance

The rendering performance depends on a number of factors such as size of the data set, viewing position, viewing direction, zoom factor, and also viewport size. Figure 10 shows one image for each of

Name	grid type	number of cells
Blunt Fin	curvilinear	37,479
Liquid Oxygen Post	curvilinear	102,675
Delta Wing	curvilinear	201,135
Neghip	rectilinear	250,047

Table 2: Datasets used for validation and performance measurements.

the data sets that were presented in Table 2. The respective detailed rendering statistics to generate those images are shown in Table 1. For all curvilinear grids, the rendering times are below 1.5 secs but the actual balance between GPU and CPU varies strongly. For the delta data set most work is performed on the GPU but for the bluntfin and post data set the CPU takes around 40% of the overall rendering time. This is due to the over-sampling that is performed in large cells so that somewhat uniform maximum sample distances are guaranteed (see Section 3.4). Generally, the time the CPU is needed in order to decide which triangles to render in the next step is negligible. In order to improve the performance of the CPU, one could investigate the use of pre-integration techniques so that the time consuming sampling could be avoided [6, 12].
Figure 12 illustrates the dependency of rendering time and the viewing parameters. For each data set, two different viewport sizes were used: 256×256 and 512×512. In each case we performed a full 360 degree rotation in increments of 2 degrees and measured the time of each frame and the number of cells rendered. The rotation direction is illustrated by the small wire-frame drawings underneath each figure. All three diagrams show clearly the view dependent performance. Looking at the results of the blunt and post data sets, rendering performance is best for straight views onto the data set. In these views, the maximum number of traversed cells is smallest, requiring the least number of iterations. For the other views, the number of traversed cells increases and requires more iterations. For a rectilinear grid, the number of iterations is equal to the maximum Manhattan distance. It takes 184 iterations to render the neghip (64^3 cells) for an almost diagonal view (Figure 10(d)). With the number of iterations, the number of actual cells processed also increases. A cell might have to be rendered more than once due to blocking rays, as indicated by the green rays in Figure 5(d). The viewport size is impacting the actual time it takes to fill the triangles and with increasing viewport sizes, the overall rendering time increases, too, see Figure 12(a), (c), and (e). However, the increase in rendering time is not only due to the fill-rate but also due to less aliasing when using larger viewports. Figure 12(b), (d), and (f) show an increased number of rendered cells. For smaller viewports and in areas with many small cells, not all cells will be able to leave a trace in at least one pixel. Hence, no ray will be traversing these cells, resulting in fewer cells traversed for smaller viewports. One way to address this could be using view-dependent grid resolutions but we have not yet looked into this.
Last but not least, the peeling algorithm performance considerably depends on the memory transfer between CPU and GPU.

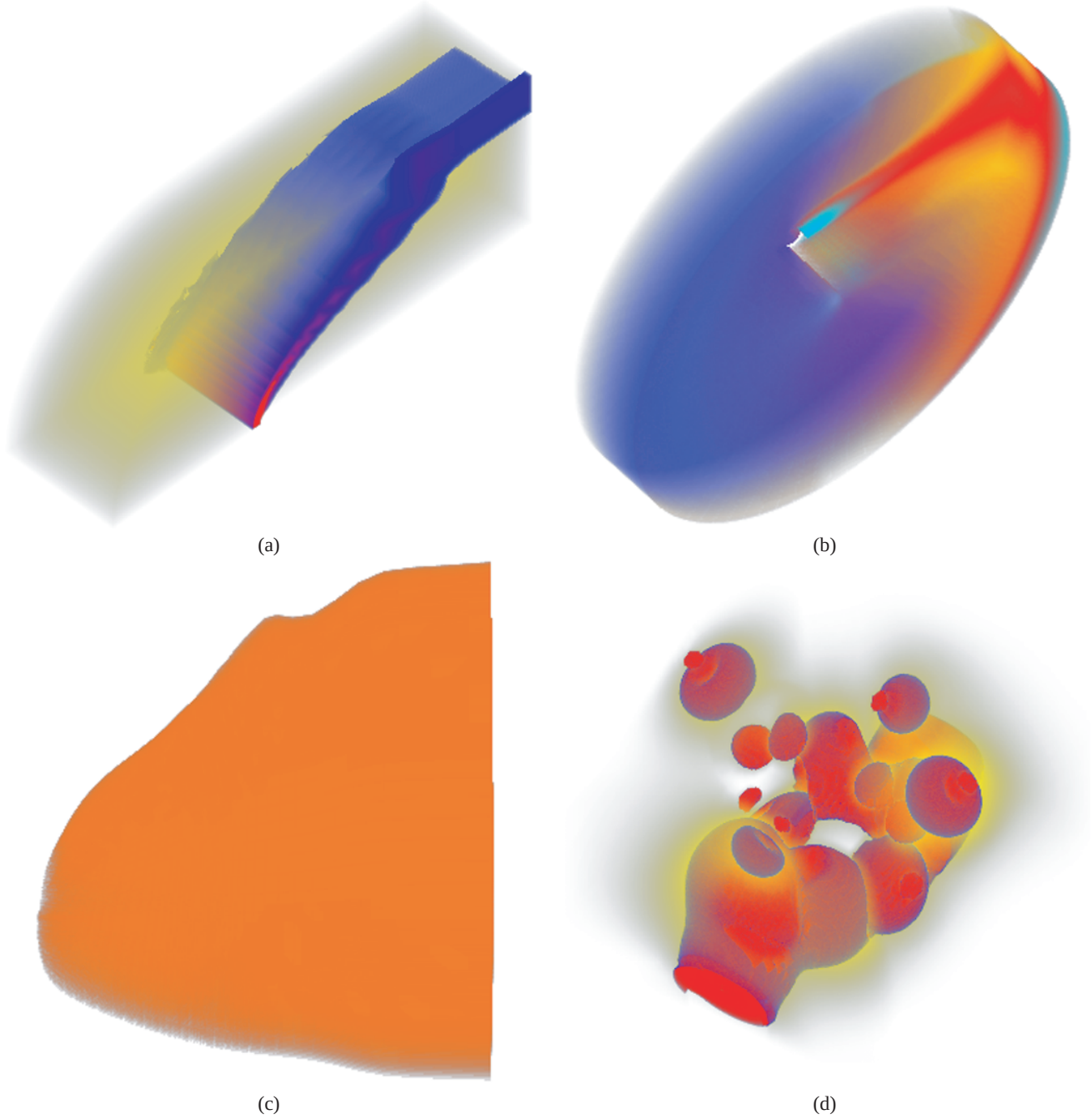

Figure 10: Volume renderings with our method: (a) Blunt Fin, (b) Liquid Oxygen Post, (c) Delta Wing, (d) Neghip.

After evaluating a couple of transfer formats, we decided to use NVIDIA's OpenGL extension to allocate directly suitable AGP memory for best data transfer and use BGRA format for reading instead of RGBA. These two optimizations accelerate the reads by approximately 20-30%, compared to using RGBA and conventual GPU to CPU transfer.

4.2 Discussion

Apart from the obvious factors influencing the performance, our algorithm relies on the accuracy of the graphics card. Due to their nature, curvilinear grids frequently contain cells that extremely vary in size. Figure 11 (a) and (b) shows this for the blunt data set. As a result, the z-buffer resolution might lead to numerical inaccuracies for tiny cells, resulting in cell entry and exit points with the same depth value and subsequently yielding to a zero integration of such cells. While this is an inherent limitation of the algorithm depending on the GPU and its precision, we reduce the impact of this by using the size of the smallest cell as minimum depth value. Typically, these small cells are classified with high opacity and in this case we still get a contribution.
Generally, our approach uses fixed point arithmetic for the cell in-

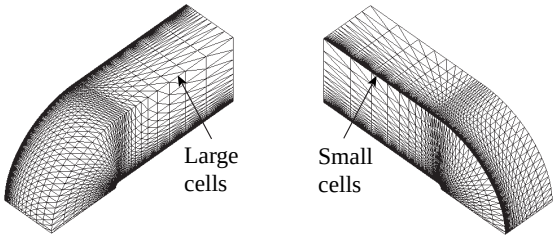

Figure 11: Blunt fin data set: areas of small and large cells.

tersections (GPU) and floating point arithmetic for the volumetric integration (CPU) while most other approaches perform sorting on the CPU and accumulate on the GPU. Hence, those approaches suffer from the limited precision of the RGBA accumulation in the GPU which has accuracy problems when using opacity weighted colors[22].

5 Conclusions

We presented a rendering algorithm for curvilinear grids based on an iterative parallel peeling approach utilizing GPU. The GPU is used for iteratively traversing the curvilinear grid (peeling) and for bilinear interpolation of the cell face. Hereby no costly preprocessing is necessary and only the actual volume integration is performed on the CPU. No special OpenGL extension is required which makes this algorithm suitable for any commodity PC equipped with a graphics card which supports standard OpenGL or equivalent instructions. Our novel algorithm produces high quality images at good performance and can easily be implemented.
The current limitations of the system are the memory transfer between GPU and CPU but since those have been improving rapidly, our algorithm will automatically benefit if this development continues. Furthermore, since GPU performance still increases at a much higher pace than CPU performance, this approach will accelerate faster over time than CPU based sorting approaches.
While our current implementation is handling curvilinear grids only, it could be extended to irregular grids, using tetrahedra instead of cubic grid cells. In addition to the current implementation, one would need to maintain a data structure for the cell neighborhood information in main memory but be able to render any type of irregular grids.

6 Acknowledgements

The work presented in this publication has been funded by the ADAPT project (FFF-804544). ADAPT is supported by *Tiani Medgraph*, Vienna (http://www.tiani.com), and the *Forschungsförderungsfonds für die gewerbliche Wirtschaft*, Austria. See http://www.cg.tuwien.ac.at/research/vis/adapt for further information on this project.
We would like to thank NASA and www.volvis.org for the available datasets. Furthermore, we would like to thank Lichan Hong and Dirk Bartz for early discussions about this approach back in 1999.

References

[1] Paul Bunyk, Arie Kaufman, and Claudio T. Silva. Simple, fast, and robust ray casting of irregular grids. In *Scientific Visualization Converence (dagstuhl)*, page 30, 1997.
[2] Richard Cook, Nelson Max, Claudio T. Silva, and Peter L. Williams. Image-space visibility ordering for cell projection volume rendering of unstructured data. *Transaction on Visualization (to appear)*.
[3] Mark de Berg, Mark Overmars, and Otfried Schwarzkopf. Computing and verifying depth orders. In *SIAM Journal in Computing*, pages 437–446, 1994.
[4] Ricardo Farias, Joseph S. B. Mitchell, and Claudio T. Silva. Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In *Symposium on Volume Visualization*, pages 91–99, 2000.
[5] Christopher Giertsen. Volume visualization of sparse irregular meshes. *Computer Graphics and Applications*, 12(2):40–48, 1992.
[6] Stefan Guthe, Stefan Roettger, Andreas Schieber, Wolfgang Strasser, and Thomasl Ertl. High-quality unstructured volume rendering on the pc platform. In *Workshop On Graphics Hardware*, pages 119 – 125, 2002.
[7] Lichan Hong and Arie Kaufman. Accelerated ray-casting for curvilinear volumes. In *Visualization*, pages 247–253, 1998.
[8] Lichan Hong and Arie Kaufman. Fast projection-based ray-casting algorithm for rendering curvilinear volumes. In *Transaction on Visualization and Computer Grpahics*, pages 322–332, 1999.
[9] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions of Graphics*, pages 245–261, 1990.
[10] N. Max, P. Williams, C. Silva, and R. Cook. Volume rendering for curvilinear and unstructured grids. In *CGI*, 2003.
[11] Nelson Max. Optical models for direct volume rendering. In *Transaction on Visualization and Computer Graphics*, pages 99–108, 1995.
[12] Stefan Roettger and Thomasl Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *Symposium on Volume Visualization and Graphics*, pages 23–28, 2002.
[13] Peter Shirley and Allan Tuchman. A polygonal approximation for direct scalar volume rendering. In *Workshop on Volume Visualization*, pages 63–70, 1990.
[14] Claudio T. Silva. *Parallel Volume Rendering of irregular grids*. PhD thesis, State University of New York at Stony Brook, 1996.
[15] Claudio T. Silva, Joseph S. B. Mitchel, and Arie Kaufman. Fast rendering of irregular grids. In *Symposium on Volume Visualization*, pages 15–23, 1996.
[16] Claudio T. Silva, Joseph S. B. Mitchel, and Peter L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *Symposium on Volume Visualization*, pages 87–94, 1998.
[17] Clifford Stein, Barry Becker, and Nelson Max. Sorting and hardware assisted rendering for volume visualization. In *Symposium on Volume Visualization*, pages 83–90, 1994.
[18] Manfred Weiler, Martin Kraus, and Thomas Ertl. Hardware-based view-independent cell projection. In *Symposium on Volume Visualization*, pages 13–22, 2002.
[19] Rüdiger Westermann and Thomas Ertl. The vsbuffer: Visibility ordering of unstructured volume primitives by polygon drawing. In *Visualization*, pages 35–ff, 1997.
[20] Jane Wilhelms, Allen Van Gelder, Paul Tarantino, and Jonathan Gibbs. Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. In *Visualization*, pages 57–65, 1996.
[21] Peter L. Williams. Visibility ordering meshed polyhedra. *ACM Transaction on Graphics*, 11(2):37–54, 1992.
[22] Craig M. Wittenbrink, Thomas Malzbender, and Michael E. Goss. Opacity-weighted color interpolation, for volume sampling. In *Symposium on Volume Visualization*, pages 135–142, 1998.
[23] Roni Yagel, David M. Reed, Asish Law, Pe-Wen Shin, and Naeem Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *Symposium on Volume Visualization*, pages 55–ff, 1996.

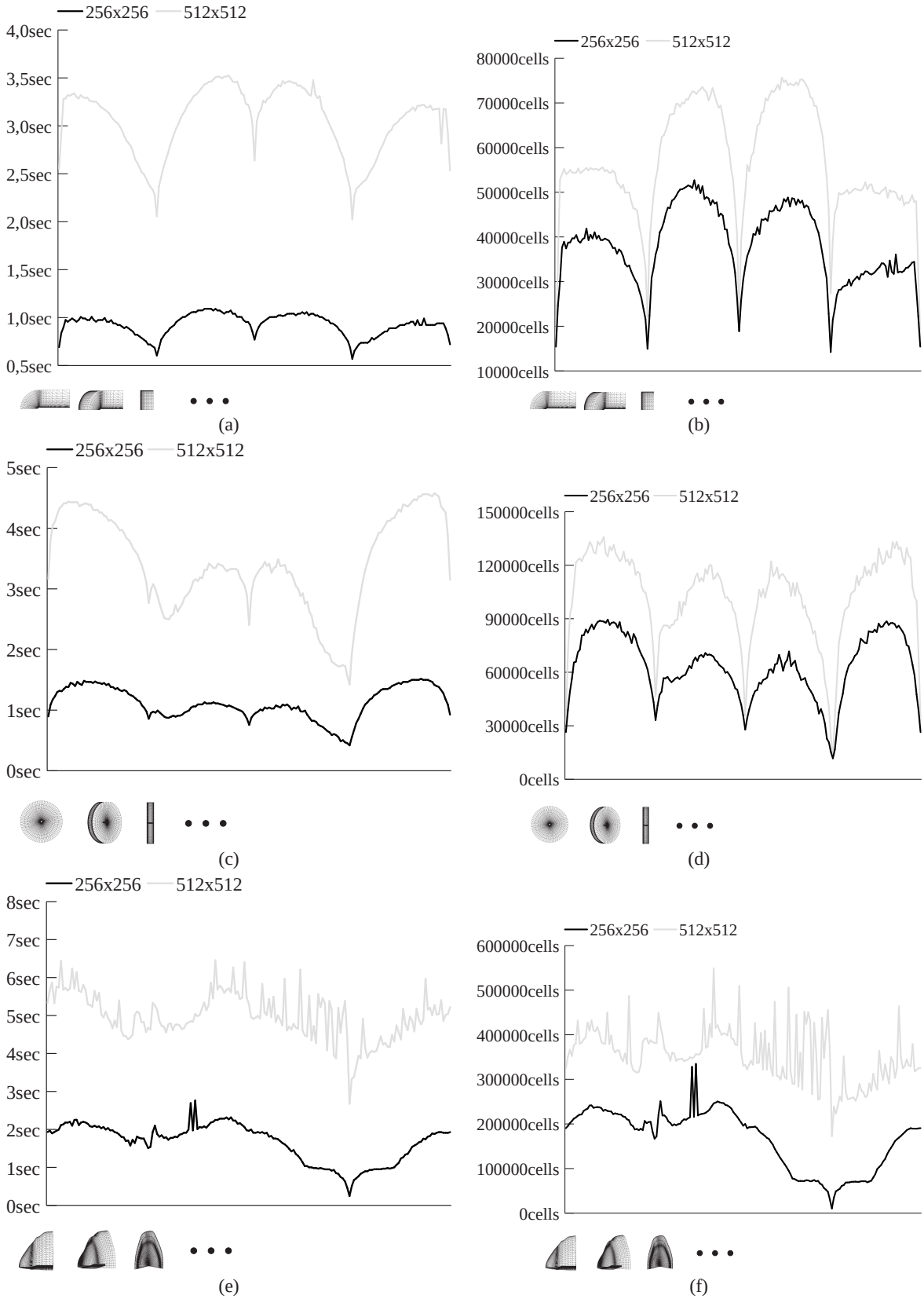

Figure 12: Render time statistics of a 360 degree rotation. Two different view-port sizes: 256^2 and 512^2. (a) and (b): Blunt fin render timings and the corresponding number of cells rendered. (c) and (d): Liquid Oxygen Post render timings and the corresponding number of cells rendered. (e) and (f): Delta Wing render timings and the corresponding number of cells rendered. Early Ray termination is enabled.