2008－AVM－63（3）
2008 / 12 / 11

Parallel Forward Projection of Large Voxel-Volumes on the GPU

Sven FORSTMANN† Jun OHYA‡

†, ‡ Waseda University, GITS Faculty, 1011 Okuboyama Nishi-Tomida Honjo-shi Saitama 367-0035 Japan
E-mail: † svenforstmann@yahoo.co.jp, ‡ ohya@waseda.jp

Abstract We present an efficient SIMD optimized implementation of the parallel voxel-based forward projection algorithm. The algorithm rasterizes RLE encoded volume data in a front to back manner by utilizing a specialized view transform. In addition to the original method, our implementation achieves a significant speedup by utilizing a multi-segment culling mechanism in combination with a binary visibility map. We show that it is possible to uncompress and visualize large volume data directly from GPU memory without streaming data from slower CPU memory. .

Keywords Volume data, Ray-casting, View-Transform, Run-Length-Encoding

1. Introduction

Recent development of SIMD graphic card architectures has made big steps and allows almost arbitrary complex algorithms to be executed on the GPU. This means that polygons as the most efficient surface representation and rendering method might be supplemented by further technologies. In this paper, we want to investigate in particular the usage and efficiency of voxels as surface rendering method. In the recent development, polygonal models are becoming more and more complex, which automatically leads to more dense meshes, where every polygon occupies just a couple of pixels on the screen. This means that voxel or point-based methods steadily gain importance as they get closer and closer to polygon based rendering. An advantage of voxels in particular is the way of modeling. Unlike polygons that require a skilled artist which knows how to handle polygon creation functions such as extrusion well, voxels allow a straightforward and intuitive way to create 3D content by painting in 3D. Especially in case of organic objects they are superior to polygons.

Challenges for the voxel-based approach are primarily memory efficiency in case of high-resolution volume data but also rendering speed as well. As basic algorithm we decided to utilize the voxel-based, forward projection algorithm [1]. The method has proven its efficiency in various software projects [2], [7] and provides advantages over related methods such as Shear-Warp [3]. It allows storing reasonably large and complex volume-data in a run-length-encoded (RLE) manner in memory by uncompressing it on the fly for rendering. According to the survey on lossless volume compression methods [8], RLE is the second fastest decoding method for uncompressing volume data. It might further be adapted to streaming from hard disk or network, but this is out of the scope of this paper.

The voxel-based forward projection algorithm, which we focus on in this paper, basically consists of two parts:

- Rasterizing RLE segments to a temporary buffer
- Map the temporary buffer onto the screen

As for the volume data, it is vertically encoded from top to bottom, leading to vertical line segments in worldspace. In run-time, these segments are uncompressed and rasterized from top to bottom and front to back.

In this paper, we want to put our emphasis especially on a parallel SIMD adaptation for recent graphics cards, beside further optimizations targeting the RLE structure.

2. Related Work

The voxel-based forward projection algorithm [1] is related to the well-known shear-warp algorithm [3]. Both algorithms render RLE volume data in a front to back manner to a temporary buffer and use a mapping technique to display it on the screen.

The major advantage of the forward projection algorithm [1] is however, that it does not need to have three copies of the volume data simultaneously in memory. Furthermore, in case of the shear warp algorithm, it is necessary to split the screen up into three viewing-planes to achieve a perspective first-person view.

Another related field are volume raytracing methods. They are different from our used forward projection algorithm since they work in the opposite way. The forward projection algorithm rasterizes all RLE voxel segments from front to back (similar to a polygon rasterizer), while a raytracer seeks the voxel that projects onto a particular pixel. Besides simple raytracing, where

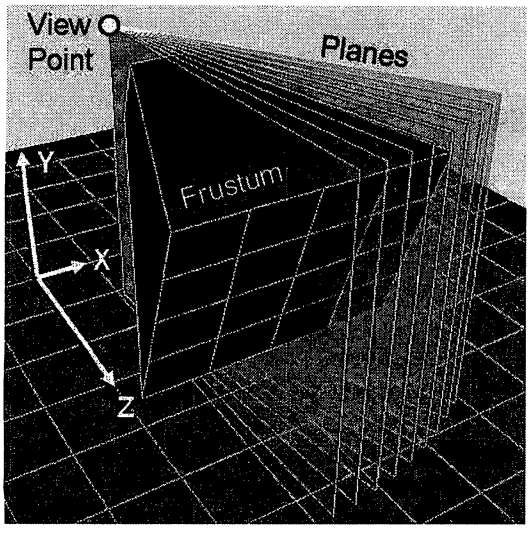

Fig.1: We raycast the scene in planes perpendicular to the y-axis of the world-coordinate-system.

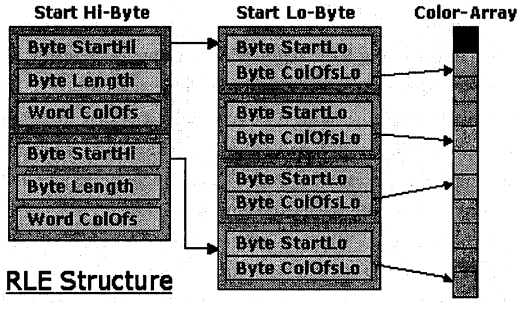

Fig.2: The hierarchic RLE structure. The first layer stores the hi-bytes of each start-address while the second layer stores the low-bytes. Total: Start = StartHi * 256+StartLo

each volume element is visited, several acceleration structures have been developed to improve performance and memory consumption.

The most common acceleration structures include octrees [13], KD-trees, nested regular grids and bounding volume hierarchies. Especially pointer-less-octrees are suitable for storing voxels very memory efficient. However, they do not allow a fast access, which is the reason why they are not used for voxel-based raytracing. More popular are pointer-based octrees. They allow fast accesses but consume significantly more memory than pointer-less octrees. The Interactive Gigavoxels algorithm [11] also is related to our method since it focuses on visualizing large voxelized surfaces on the GPU as well.

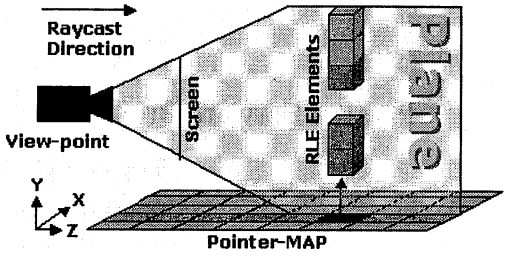

Fig.3: Per plane raycasting: In each plane we render the RLE elements from front to back and top to bottom.

Their algorithm uses nested regular grids for raycasting and achieves a performance comparable to ours.

A different group of related algorithms are point-based rendering algorithms. One of the most popular in this context is called Qsplat[9], which inspired many other researchers to come up with similar point based surface rendering approaches. An evolution of Qsplat is FarVoxels [10], as they improve on the basic point-based-rendering idea by combining it with polygonal rendering.

3. The proposed Algorithm

The voxel forward projection algorithm raycasts the scene in planes, as shown in Fig.1. Each plane starts in the view-point and progresses perpendicular to the world-space's y-axis. We define the world-space's x- and z-axis as the base of the horizontal plane, while the y-axis points upward. Each of the planes intersects the view frustum and is visible on the screen as one line. The amount of planes has to be adjusted dynamically to achieve an optimal coverage of the pixels on the screen.

The raycasting is done in a front to back manner. For each of the planes, one ray is casted starting from the view-point and ending at the maximum view distance. In each position of the ray, all RLE elements are rendered as line-segments from top to bottom.

3.1. Structure of the RLE volume data

For the RLE structure, multiple types are available to solve our task. The most simple is to stores two bytes for each element, where the first byte defines the number of skipped voxels and the second byte the number of following opaque voxels. This allows encoding arbitrary sized structures since the numbers are relative to the previous offset. However, accessing a single element requires uncompressing all previous elements, which is not very efficient. We therefore decided to use a two-level structure that allows binary searches for identifying one element in O(log n). Such queries are very important in

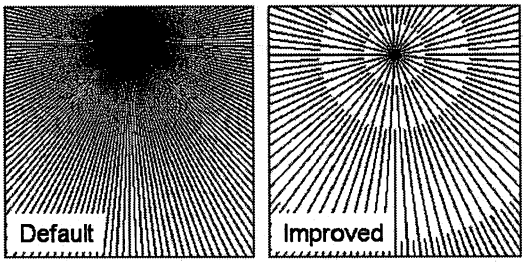

Fig. 4: Plane distribution: Looking downwards leads to the upper plane distribution on the screen. We readjust the planes (left to right) to reduce the overdraw and the raycasting cost significantly.

case of collision detection e.g.

The detailed structure that we use can be seen in Fig.2. We utilize two levels to encode columns with up to 2^{16} in size. The first level stores the Hi-bytes of the start address while the second level stores the low bytes of the 16 bit start address. The run-length of each RLE element can be computed as the difference between the actual RLE element's color offset and the previous element's color offset. The color offset is split in two parts: The 16-bit "Word" offset in the left column and the 8 bit offset in the right column which is added to the 16 bit offset for the final address.

3.2. Computing the Planes

Our algorithm raycasts in planes perpendicular to the world coordinate system's y-axis as in Fig.3. All planes are originated at the viewpoint. As a first step, we need to compute the number of planes and their parameters, which is very different from a conventional ray-tracer. There, simply one ray is casted for each pixel on the screen. In case of our method, the number of planes depends on the camera's view-angle. In case of looking straight forward, we need to have as many planes as there are columns on the screen - the number increases however if we tilt the camera downwards to about four times, see Fig.4. We therefore readjust the planes according to a fixed pattern, which leads to a much better performance without significant changes in quality.

For each of the planes, we need to compute certain parameters, such as the 2D-intersection-coordinates with the screen and the x-z-angle.

3.3. Accurate Grid traversal

We apply an accurate grid traversal as described in [4] that takes care of each cube intersection and hence leads to a correct rendering where each voxel is rasterized as cube. The accurate traversal is slightly slower than the

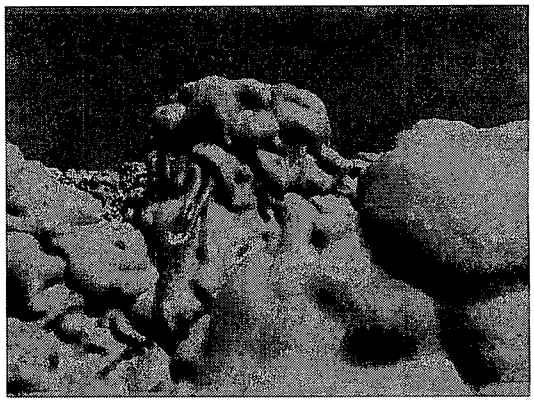

Fig.5: The depth-buffer can successfully be utilized to compute normal vectors on the fly.

equidistant, but leads to much better visual results.

3.4. Rasterizing the RLE Elements

The main part of the method is the raycasting process that visualizes the RLE elements. To achieve this, we start with the first plane at the viewpoints position and traverse all RLE columns from front to back and top to bottom. Each rendered plane maps to a line on the screen. As the lines are in arbitrary direction, we render each line into an array that is mapped to the screen as a post-process.

Computation wise, we need multiple operations per RLE element: First the translation to the camera position, then the rotation according the camera tilt and finally the perspective divide.

The rotation around the y-axis (panning) is automatically achieved based on the plane's orientation. Last missing is the basic 2D rotation around the screen center to create a full 6 DOF camera.

Visibility Culling

We include multiple culling mechanisms in order to speed up the rendering. The first culling mechanism is the floating horizon algorithm [5], which already speeds up the rendering significantly. This algorithm has the advantage in combination with RLE that we can skip most of the elements for certain scenes. Especially landscape scenes suit very well. There, it is often sufficient to only test the most upper RLE element to know if all other elements of the column can be skipped.

In addition to floating horizon, we further utilize the largest drawn line segment that does not touch any boundary for culling. In case another drawn line segment partially covers the previously largest, both are merged together, increasing the cull-able area.

Fig.6: The sample scene consists of a repeatedly rendered 1024^3 dataset and contains numerous complex shapes in order to provide an accurate measurement.

This approach is different from the Shear-Warp algorithm [3], which uses a forwarding buffer to cull invisible segments. However, it is possible to combine both of these methods efficiently.

3.5. Optimizations

In order to speed up the rendering, we add two further modifications. The first is to utilize the GPU's shared memory to perform a quick visibility check. The shared memory size of our graphics card has been 16KB, which is sufficient to store one bit for each pixel on the screen to prevent overdraw and save memory bandwidth. In case of 128 parallel threads and a maximum screen resolution of 1024, we need exactly 128*1024/8 = 16KB of memory.

The second optimization is to switch from an accurate grid traversal to an equidistant traversal for distant geometry. This leads to a better performance as it reduces the amount of divergent branches. For SIMD architectures, divergent branches are one of the most serious problems.

Dataset	Optim.	Far View	Near View	Avg	Version	Screen
Full	F,C	43	45	44	GPU	512x512
Full	S,C	43	42	42.5	GPU	512x512
Full	F,C	17.5	21.2	19.35	GPU	1024x768
Full	S,C	19.2	19	19.1	GPU	1024x768
Surface	S	18.8	18.5	18.65	GPU	1024x768
Surface	S,C	18.2	17.8	18	GPU	1024x768
Full	S	16.3	14.7	15.5	GPU	1024x768
Surface	C	15.3	15.6	15.45	GPU	1024x768
Surface	F,C	15	15.8	15.4	GPU	1024x768
Surface	F	14.9	15.8	15.35	GPU	1024x768
Full	C	15.4	15.3	15.35	GPU	1024x768
Surface		14.7	15.1	14.9	GPU	1024x768
Full	F	13.9	15.6	14.75	GPU	1024x768
Full		10.8	10.2	10.5	GPU	1024x768
Full	C	1.7	2.1	1.9	CPU	1024x768
Full		1.5	2	1.75	CPU	1024x768
Surface	C	1.2	1.6	1.4	CPU	1024x768
Surface		1.2	1.4	1.3	CPU	1024x768
Optimizations: (S)haremem, (C)entersegment, (F)orward						

Table 1.Performance: The speed results are in frames per second (fps) for the near and far view of Fig.6. We compared different optimizations and two different encodings of the same dataset (Full=solid, Surface=only surface voxels). Avg represents the average fps of near and far, version indicates whether we used GPU or CPU and screen states the used screen resolution.

3.6. Mapping the Temporary Buffer

We can draw the temporary buffer efficiently onto the screen in parallel as textured lines. Keeping in mind the previous rendering process, four different texture directions are utilized based on the ray-cast direction.

3.7. Closing Gaps

The gaps that still exist on the screen are closed by pixel repetition, which leads to a result comparable to texture mapping. Texture mapping cannot easily be used directly, as the temporary buffer contains columns of variable size. This efficiently reduces overdraw but complicates the mapping to the screen.

Filling the gaps has to be done in horizontal and vertical direction separately due to the concentric way we are mapping the temporary buffer on the screen.

3.8. Implementation Details

The algorithm is written in C++ and uses CUDA for SIMD rendering on the GPU. Equal to the original method, also our implementation utilizes volume-data mip-maps to speed up the rendering of distant RLE elements. The SIMD parallelization of the method is achieved by assigning one plane to each GPU core.

The shading of the Richtmyer-Meshkov dataset is done by screen space normals (SSN). They are calculated from the depth buffer by the pixel-shader in one pass as demonstrated in Fig.5. Together with screen-space-

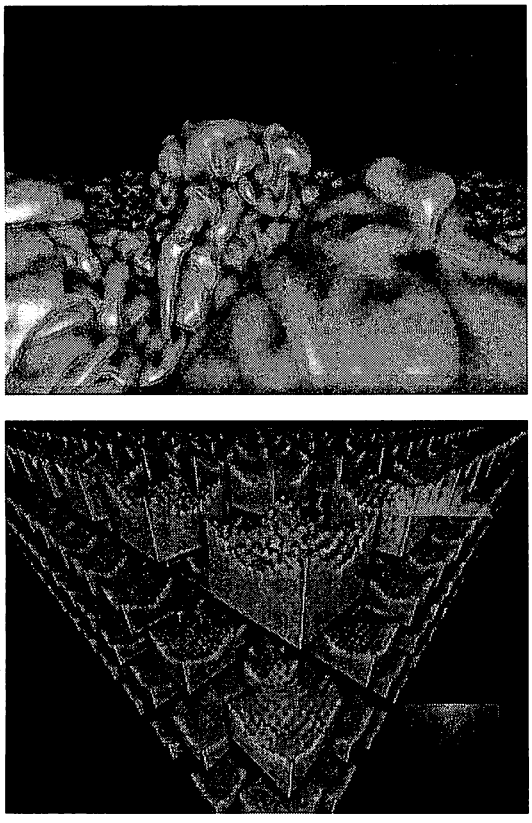

Fig.7: It is even possible to render the complete dataset more than 100 times at interactive rates at a 1024x768 screen resolution. For the shading, we utilized a combination of screen-space-ambient-occlusion and screen-space normals.

ambient-occlusions (SSAO) [12], the shading is satisfying and results in a good depth perception. Since the required gradient for the normal computation needs a certain area, the positive effect of SSN is an adjustable region for the normal smoothing – however, sharp edges cannot be shaded easily which leads to false results in certain cases.

4. Contributions

Our proposed modifications contribute to the existing algorithm as follows:

- Our two-level RLE not only allows encoding large volume-sizes with an average of 1 byte per surface voxel, but also supports binary searches that are useful for raytracing or collision queries.
- Our implementation is able to store a binary version of the Richtmyer-Meshkov instability data-set's iso-surface (2048^3 voxel) in GPU memory and render multiple instances shaded on a consumer level GPU with just 256 MB of video ram. We achieve interactive frame rates and don't requiring any streaming from hard-disk.
- Our three-way visibility culling algorithm significantly speeds up the initial method
- Unlike the original method, our algorithm applies a better distribution of the planes on the screen which increases the speed up to 50%.
- Our implementation takes advantage of the parallel SIMD GPU architecture by using CUDA. This speeds up the method about 5x compared to the CPU implementation as it is possible to run the raycasting process in parallel where each plane is handled by one of the GPU's processing units.

5. Limitations

Besides the advantages of the presented method, it also has several limitations.

- Since the number of planes depends on the camera tilt, the speed is not constant. In case of common terrain-like out-door scenes this is hardly noticeable, but in case of complex scenes such as the Richtmyer-Meshkov instability dataset we observed a performance impact.
- Three-dimensional voxel filtering to smooth voxels close to the camera is not trivial since all neighboring elements are run-length-encoded.
- Anti-aliasing can be achieved by increasing the screen resolution and re-sampling the output at a lower resolution. However, it is difficult to directly perform anti-aliasing on certain selected edges.

6. Experiments

We conducted results using two different scenes with different resolutions for rendering as well as different resolutions for the voxel volume. Our testing hardware has been a Pentium-D 3.0Ghz Processor with 1 GB of RAM and a Geforce 8800 GTS (384MB) graphics board.

The first test-scene is shown in Fig 6. It's an artificially generated scene with a resolution of 1024^3 that contains color information for each RLE element. Our second scene for testing is the well-known Richtmyer-Meshkov dataset with a resolution of 2048^3 at time step 219 and at iso-value 60. Since its surface is very complex, we were only able to store the geometry data itself in GPU memory (198MB), but not normal vectors or color data. In order to still shade the rendered geometry, we utilize SSAO and SSN in combination, which requires about 2ms for the post-processing at the used 1024x768 resolution of the screen.

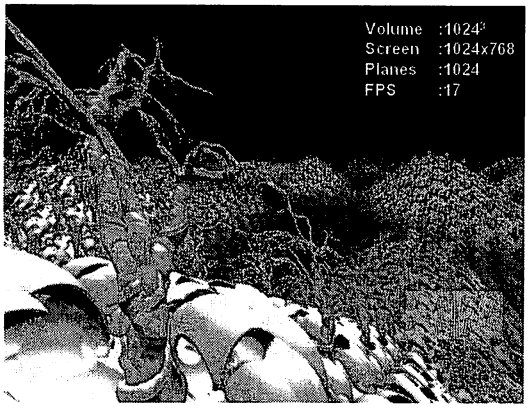

Fig.8: Landscape: For this terrain we set the view-distance to 180.000 voxel. It renders well at about 15-30 fps at a 1024x768 resolution.

As test-scene, we repeatedly rendered the 1024^3 dataset shown in Fig 6 to cover an area of about 40.000 x 1024 x 40.000 voxels. The screen resolution for testing has been 1024x768, the number of planes 1024. The detailed results for this scene can be seen in Table 1. We combined various optimizations to figure out which one fits best. It turned out, that using our proposed culling (centersegment) in combination with the forward-skipping (forward), described in the Shear-Warp algorithm, works best. Using the GPU's shared memory for visibility testing also worked very well and achieved results comparable to the forward skipping. The optimization that readjusts the planes on the screen as in Fig.4 led to a speedup of up to 50% when looking straight downwards.

For rendering a single instance of the Richtmyer-Meshkov dataset, we achieved at average 16 fps within a range of 9 to 40. In case of rendering the dataset repeatedly to create a virtual 40k x 2k x 40k resolution, we achieved about 10 fps in average with a range of 8 to 40 fps. The heaviest of all scenes can be seen in Fig.7. It contains about 400 copies of the dataset and hence only rendered with 5-8 fps on average. The slow performance is mostly due to the algorithm itself. It is well suited for hilly terrain scenes such as in Fig.8 where the ground is covered with objects. There, the floating horizon algorithm culls away most geometry. In arbitrary scenes however, the culling is not as efficient and hence the performance is not as high.

In total, our SIMD GPU implementation achieved a speedup of about 5x compared to the CPU based implementation shown in Table 1.

7. Conclusion

We have presented an efficient adaptation of the voxel forward projection algorithm to recent SIMD graphics hardware by further proposing several modifications to speed up the algorithm and optimize memory accesses. The proposed algorithm is about 10x faster than the basic CPU implementation and even able to visualize the full resolution Richtmyer-Meshkov dataset shaded on a single NVidia GTS 8800 GPU at interactive frame-rates without any streaming from CPU memory or hard-drive.

References

[1] John R. Wright and Julia C. L. Hsieh, "A voxel-based, forward projection algorithm for rendering surface and volumetric data", Visualization'92, pp.340--348, 1992

[2] Ken Silverman, Voxlap engine, 1999-2003, http://advsys.net/ken/voxlap.htm

[3] Philippe Gilbert Lacroute, "Fast volume rendering using a shear-warp factorization of the viewing transformation", SIGGRAPH '94, pp.451--458, 1994.

[4] John Amanatides, Andrew Woo : "A fast voxel traversal algorithm for ray tracing", Eurographics '87, pp.3-10, North-Holland, 1987

[5] T.J. Wright, "A Two-Space Solution to the Hidden Line Problem for Plotting Functions of Two Variables," IEEE Trans. Computers, vol. 22, no. 1, pp. 28-33, Jan. 1973.

[6] NVidia Corp, Compute Unified Device Architecture (CUDA) http://developer.nvidia.com/object/cuda.html

[7] Visualization Lab, Center for Visual Computing, SUNY Stony Brook: Voxel-Based Flight Simulation http://www.cs.sunysb.edu/~vislab/projects/flight/

[8] Philippe Komma and Jan Fischer and Frank Duffner and Dirk Bartz: "Lossless Volume Data Compression Schemes, SimVis 2007, pp. 169-182, 2007

[9] Szymon Rusinkiewicz and Marc Levoy:" QSplat: a multiresolution point rendering system for large meshes", SIGGRAPH '00, pp. 343—352, 2000

[10] Enrico Gobbetti and Fabio Marton:"Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms",SIGGRAPH '05, pp. 878--885,2005

[11] Crassin, Cyril and Neyret, Fabrice and Lefebvre, Sylvain:"Interactive GigaVoxels",INRIA Technical Report,June 2008

[12] Martin Mittring: "Advanced Real-Time Rendering", 3D Graphics and Games Course, Chapter 8, pp.113-115, SIGGRAPH 2007

[13] Aaron Knoll, Ingo Wald, Steven Parker, Charles Hansen:"Interactive Isosurface Ray Tracing of Large Octree Volumes", IEEE Symposium on Interactive Ray Tracing, pp.115-124, 2006