Visualizing run-length-encoded volume data on modern GPUs

Sven FORSTMANN† and Jun OHYA‡

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

Abstract We present an algorithm for the visualization of large opaque volume data. So far, volume data has mainly been visualized on the GPU using conventional ray-casting or texture-based techniques. However, since the appearance of CUDA, even more complex visualization techniques become possible. We show that large run-length-encoded volume data can efficiently be visualized using CUDA. In our experiments using a NVIDIA 8600 GTS graphics board, even as much as 40.000 x 1024 x 40.000 voxel were visualized in real-time at interactive 40 fps.

Keyword Volume data, run-length-encoding, ray-casting

1. Introduction

Volume data visualization is an important topic in the area of computer graphics, especially for the visualization of data captured from medical imaging, such as data from MRI or CT. However, volume data is also interesting for the visualization of surface data, such as converted from polygons. Volume data from polygons has in particular the advantage that it can be compressed very efficient, since most of the volume pixels (voxels) are empty, which makes it very feasible for compression algorithms such as run-length-encoding (RLE).

In history, many algorithms have been invented to solve the task of volume rendering efficiently. The most common algorithms can generally be divided into the following categories: ray-tracing, ray-casting, texture based volume rendering and finally rendering iso-surface rendering. In our research, we would just like to focus on ray-casting based volume rendering. However, we will also briefly introduce other common algorithms.

1.1. Related Work

The easiest way to visualize volume data is to shoot a ray for each pixel of the view-plane and trace along until a non-empty voxel has been reached. This method is the easiest, but at the same time computationally the most expensive as it requires an effort of $O(n^3)$. In this case, usually regular grids or tree-structures help to accelerate the tracing process. A paper describing a generic ray-tracing algorithm is "A Fast Voxel Traversal Algorithm for Ray Tracing" [1]. There, the focus was put especially on optimizing the traversal algorithm to trace grid-intersections efficiently and accurately without much effort.

A more advanced algorithm is the famous "Shear-Warp" algorithm of Lacroute et.al.[2] , which still belongs to the fastest methods for visualizing volume data. It basically utilizes three RLE encoded copies of the original volume data, which are traversed from front to back in an isometric manner. The main advantage of this algorithm is that rays are always shot from front to back in object space rather than in screen space. This speeds up the traversal process significantly but requires a final view-transformation (warp) of the rendered image to obtain the correct result. Due to the RLE compression and the algorithm behavior to traverse from front to back in object-space, thee copies of the volume data are necessary as front to back might be either in x-,y- or z-direction. The algorithm is also capable of ray-casting the scene in a perspective manner, involving a slight modification.

Another possibility to visualize volume data is the so-called iso-surface. It is usually used along with the marching cubes algorithm and in particular useful to clearly visualize bone or skin structures of medical data as polygonal objects.

The last most frequently used method to visualize semi-transparent volume-data is conventional texture-mapping, as it is available on any recent graphics card. For the visualization, the volume data is simply rendered as textured polygons, where each layer of the volume data is represented by one polygon.

A comprehensive overview of all briefly introduced methods might be found in Todd et al, "A survey of algorithms for volume visualization" [3].

2. Algorithm Design

We plan to implement the so far almost undocumented

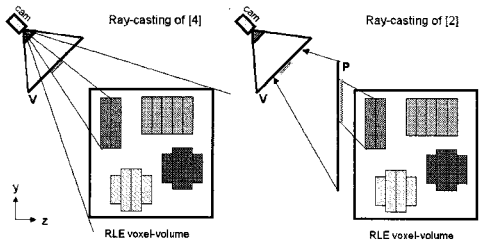

Fig.1.System comparison between [2] and [4]. V represents the view-plane while P represents a virtual projection plane that is mapped to V.

algorithm of the VoxLap engine by Ken Silverman [4].

The algorithm is related to the "Shear-Warp" algorithm of Lacroute et.al.[2], as it utilizes ray-casting the a RLE-encoded volume data. However, the major difference between the two algorithms is that the original Shear-Warp method requires three copies of the volume data that is to be visualized, since the warp-method is not very flexible. In VoxLap, a different warping is being used, which allows visualizing the scene with only one copy of the volume data. However, as the mapping technology is more complex, the plain algorithm is not as fast as the original one proposed by Lacroute et.al.[2].

We therefore intend to improve the ray-casting mechanism in various ways, to achieve a better performance also by employing additional culling algorithms.

As recent GPU's do not only support shader languages but also allow the execution of conventional c-program code, a further intention of our research is to exploit the performance gain from CPU to GPU. Since older GPU versions do no support arbitrary c-programs, also the algorithms that might be implemented are very limited. Our intended method cannot be implemented using plain shader languages (such as GLSL), since random writes to the memory first became available with CUDA [5].

3. Contributions

The main contribution of our research is the efficient GPU implementation of [4] beside further culling optimizations to speed up the algorithm, the comparison between GPU and CPU and the first document to describe the implementation of [4] in detail.

Our initial GPU implementation is able to visualize an area of up to 40.000 x 1024 x 40.000 voxels at interactive 30 frames per second on a GeForce 8600 GTS graphics board.

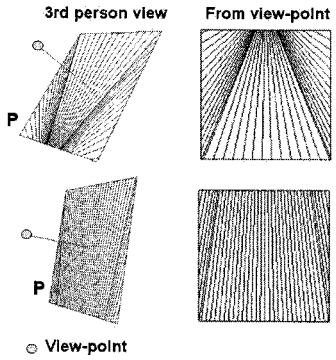

Fig.2.Raymapping. We can see that the scene needs to be rendered in columns, which are oriented to the direction to the RLE, rather than the view-plane P.

4. The algorithm

The general algorithm principle is similar to [2] and consists of two parts. The first part is the ray-casting part while the second part is the ray-mapping part. The volume data is run-length-encoded (RLE) and hence consists of numerous RLE elements. We consider one RLE element as a vertical row of opaque voxels of same color. The information that is required to be stored for each RLE element is basically the vertical offset of the element in RLE-direction (here we use the y-axis), the run-length and finally the color information. Additionally, a normal vector, certain material properties or texture coordinates might be stored along.

Other than described in [2], the presented algorithm does not ray-cast onto a virtual projection plane. It directly ray-casts to the view plane. However, in order to speed up the rendering process and avoid the artifacts from non-uniform rendering to the screen space, also here an additional mapping is involved.

4.1. Basic Ray-casting

The ray-casting of the presented algorithm is in principle very different from conventional raytracing algorithms: not one ray per pixel on the viewing-plane is shot into the scene, but one ray for each column of the view-plane (considering the observer is looking straightforward). Important at this point is also to mention that the RLE encoding direction has to be top-down (y-axis). This is different from [2], where the RLE-direction is changing depending on the actual projection plane (which might differ depending on the view plane!). To demonstrate the ray-casting more in

detail, Fig.1 is included. Inside the figure, V represents the view plane while P represents a virtual projection plane that is used in [2]. Comparing the required operations for the ray-casting process [2] requires a two-dimensional rotation and a successive perspective transformation for each RLE element, while [4] only requires a perspective transformation for each element and a consecutive mapping of the rendered projection plane onto the view-plane.

4.2. Ray Mapping

So far, we regarded the ray-casting process from a side-view in Fig.1, where we investigated the ray-casting for one slice of the volume data. However, if we take a perspective look onto the view-plane (Fig.2), how the ray-casting is been made, we can see that we get a very unique pattern. The pattern is generated due to the RLE direction, which is top-down inside Fig.2. As for the algorithm, it is not practical to ray-cast directly onto the screen in the direction of the lines, which we can see in Fig.2. The reason is, that many artifacts are generated, as there are large gaps between the lines. A better way, which is already implemented in [4], is to ray-cast each line into a temporary array, similar to generating a texture. After the ray-casting process is finished, all temporary arrays can be regarded as a texture that simply needs to be mapped onto the view-plane. The mapping might even be shifted to the graphic card's rasterizer to allow a hardware accelerated mapping of the temporary data, including automatic bilinear filtering to avoid aliasing.

A second issue regarding the ray mapping is, that the distribution of the rays on the view-plane should be averaged, meaning that each pixel should be covered by approximately one line. In our case we achieve this by firstly calculating a vanishing point VP. VP is the intersection of the view-plane with the y-axis, aligned to the viewpoint. Once this point is computed, it is possible to calculate a set of lines on the view-plane which represent the result of each ray that is casted to the volume data. The method we implemented to compute the ray-directions is straightforward. We compute the angles between the view-plane corners and create partitions in the view-plane accordingly. The ray-density is then linearly calculated based upon the angle of two neighboring corners and the vanishing point VP.

4.3. Culling

An important step in the presented algorithm is culling. It is necessary to avoid overdraw as much as possible, as otherwise the speed will drop dramatically. Other than ray tracing, where the ray might be terminated as soon as an opaque object is hit, we are required to cast a ray up to the maximum distance and test the visibility for RLE elements in each position of the ray. Therefore, algorithms helping us to allow an early ray-termination are very important. In our case, we implemented the floating horizon algorithm [6], a classical method that is applicable efficiently in our case.

Fig.3.Culling. The three steps show the overdraw reduction by adding culling. 1. No culling, 2.floating horizon 2. Floating horizon plus an additional segment.

For each column that is ray-casted, we remember the topmost and the bottommost position on the screen. Now during ray-casting, all RLE elements are step by step projected onto the view-plane, where we process them in a near to far manner. If one of the RLE elements touches the topmost or bottommost border, we may update the touched border and narrow the region that might be hit by further RLE elements. Simultaneously, we also remember the number of drawn pixels. If either both boundaries (upper and lower) touch each other, or the number of drawn pixels equals the number of total pixels in this particular column, we may skip rendering further RLE elements and hence achieve an early ray-termination. This is in particular useful of very complex scenes, where many details are visible but the view-distance is very limited due to the amount of details.

An additional step, which we employed, is, to track a third boundary region that is independent from top and bottom. This allows a further optimization of the ray-casting to limit the overdraw. We included Fig.3 to demonstrate the efficiency of the overdraw reduction.

A different optimization that has been implemented is an early skipping of RLE elements. Conventionally, it is necessary to test the visibility for all RLE elements of a given x-z-position. However, as they are sorted in y-direction, it is possible to skip all elements that are below or above the current view-plane boundaries.

The last optimization that is being implemented is a

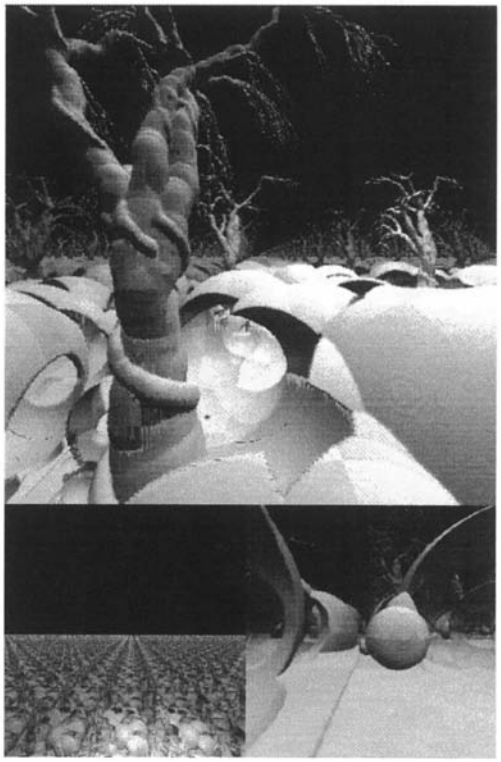

Fig 4. Volume rendering. Here we can see the first results of ray-casting a reasonably large test dataset. A cube of 1024^3 voxels is repeatedly rendered to create a scene lasting over 40k x 1k x 40k voxels. Taking a look onto the lower left image, each of the trees there is equivalent to one 1024^3 dataset.

coarse to fine approach using a regular cell-grid to speed up the ray-casting process. Similar to axis aligned bounding boxes, we firstly test the visibility for each cell, which speeds up the ray-casting process up to two times.

5. Implementation

We implemented the algorithm using the Nvidia's CUDA. The rendering can completely be done on the GPU, However, we distributed the tasks along GPU and CPU. In our implementation, the CPU is responsible for computing the line segments that have to be ray-casted, while the GPU is doing the parallel ray-cast. As the GPU is highly parallel with up to 100 processing units and more, we assigned ray-casting each column on the view-plane to a single unit. This means that it is possible to ray-cast up to 100 lines and more in parallel.

6. Results

In first tests, we were able to visualize up to 40000x1000x40000 volume pixels (voxels) at 40 frames per second. This is significantly larger than most existing methods might visualize interactively, as the limitation for common existing methods lies between 256x256x256 and 2048x2048x2048.

Comparing the amount of raw memory that is being used for the different resolutions, there are 24 MB of data for a 256^3 resolution, 8 GB of data for a 2048^3 data-set, and finally 1.5 TB of data for our 40k x 1k x 40k data-set. Considering that a conventional Desktop-PC only has about 1 GB of main memory, we can see how important data compression and level of detail is about to become.

In Fig.4, we can see the results of or preliminary version. For testing, we have created a dataset of 1024^3 voxels that we render repeatedly, similar to a texture.

For testing purpose, we also implemented dynamic lighting into the algorithm, which surprisingly didn't caused too much slow down – the frame-rate only dropped about 2-3 frames from 40 to 37.

As for the comparison between GPU and CPU, we found that the using the GPU, a speedup of 2-3x might be achieved. In our case, the testing system has been a 3.0 Ghz Pentium-D CPU (Dual-core), 1 GB ram, with an Nvidia GeForce 8600 graphics card.

As the computational power of recent graphic cards is up to six times of a common CPU, we tried to figure out the reason why the GPU-CPU difference is only 2-3 times. Detailed tests allowed us to investigate the reason: It is mostly due to branching costs. One additional branch on the GPU might result in a 33% drop of the algorithms efficiency. In our final implementation, we achieved a average GPU usage of 16%, which leaves much space for improvement and future work.

Bibliography

[1] J. Amanatides and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing", Proc. Eurographics '87, Amsterdam, The Netherlands, August 1987, pp 1-10.

[2] Philippe Lacroute and Marc Levoy: "Fast volume rendering using a shear-warp factorization of the viewing transformation", SIGGRAPH '94, pp.451—458

[3] T.Todd Elvins, "A survey of algorithms for volume visualization", Computer Graphics, vol.26-3, pp.194--201, 1992

[4] Ken Silverman's Voxlap: http://advsys.net/ken/voxlap.htm visited Jan. 2008

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

[6] 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.