Paper

Efficient, High-Quality, GPU-Based Visualization of Voxelized Surface Data with Fine and Complicated Structures

Sven Forstmann[†], ***non-menber and*** **Jun Ohya**[†], ***member***

SUMMARY This paper proposes a GPU based method that can visualize voxelized surface data with fine and complicated features, has high rendering quality at interactive frame-rates, and provides low memory consumption. The surface data is run-length-encoded (RLE) for each level-of-detail. Then, the loop for the rendering process is performed on the GPU for the position of the viewpoint at each time instant. Raycasting of the scene is done in planes, whereas each plane is perpendicular to the horizontal plane and passes through the viewpoint. For each plane, one ray is cast to rasterize all RLE elements intersecting this plane - starting from the viewpoint and ranging to the maximum view distance. The rasterization process projects each RLE element passing the occlusion test onto the screen at a level of detail that decreases with distance of the RLE element from the viewpoint. Finally, smoothing of voxels in screen-space and full-screen anti-aliasing are performed. To provide lighting calculations without storing the normal vector inside the RLE data structure, our algorithm recovers the normal vectors from the rendered scene's depth-buffer. After the viewpoint changes, the same process is re-executed for the new viewpoint. Experiments using different scenes show that the proposed algorithm is faster than the equivalent CPU implementation and other related methods. Our experiments further prove that our method is very memory efficient, and achieves high quality results.

key words: *Volume data, Voxels, Raycasting, Splatting, View-Transform, Run-Length-Encoding..*

1. Introduction

1.1 Background

Traditionally, polygon based rendering has been more efficient than point primitive (splatting) or volume-pixel (voxel) based rendering. However, polygonal models are becoming more and more detailed, leading to dense meshes where each polygon merely covers a few pixels on the screen. Once polygonal meshes become so dense, rendering results by polygons, point primitives or voxels do not show significant differences in quality and rendering speed. This means that voxel and point-based rendering methods gain more importance. This is because as the rasterized size of voxels, splats and polygons become similar, rendering a voxel or splat employs considerably less computation than rendering a polygon.
Previously, an advantage of polygon-based rendering was the ability to use repeating textures to save memory. Voxel and splat based rendering inherently use unique texturing, so there is no benefit in memory consumption from using repetitive texturing. However, as there is a recent trend to use unique, non-repeating, textures for each object on the screen (Megatexture technology [15]), the memory consumption for polygon-based rendering sharply increases. Therefore as this trend continues, the memory consumption of voxel and splat-based rendering becomes comparable with unique textured polygon rendering.

Next, we want to compare the use of level of detail (LOD) between voxels and polygons. LOD is important to accelerate the rendering and decrease the run-time memory consumption. For polygonal objects, LOD is usually implemented as follows. First, a set of polygonal objects with different levels of detail is created by the artist. Then, at run-time, the proper LOD of the object is selected according to the distance from the object to the camera and the object is visualized on the screen. Voxels can handle LOD more efficiently than polygons, because voxel data can be easily down-sampled for representations in lower details. Therefore, it is not necessary for an artist to design separate models of the same object for each level of detail, the different levels of detail can be generated automatically.
The final advantage of voxels over polygons we wish to mention is, Boolean operations can be applied much easier to voxels compared to polygons.
Despite all these advantages of voxels over polygons, it should be noted that deformations and skeletal animations in real-time still pose a challenge for voxel-based representations.
Point and voxel based rendering are very similar. However, the major difference between voxel based rendering and point-based rendering is that voxels occupy a well-defined cubic portion of volume in space, while point-based methods approximate the geometry usually by 2D splats.
For reasons related to the fact that splats are 2D approximations of a 3D object, there have to be several exceptions implemented for a point-based algorithm to be robust. Voxel based algorithms are generally more robust without the need for such exceptions, because a voxel covers a well-defined portion of space in 3D.

1.2 Related Work

We split the related methods into three groups: rendering

Manuscript received January xx, 20xx.
Manuscript revised March xx, 20xx.
†The author is with Waseda University, Tokyo, Japan.

voxel volume data by using Shear-Warp [3], raytracing based algorithms and point based rendering. We will briefly overview the most widely used methods in each group and mention their key issues.
Shear-Warp [3] renders RLE volume data in a front to back manner to a temporary texture. The temporary texture is then mapped to the screen. It has been proven to be very fast for dense, semi-transparent volume data. However, it requires storing three copies of the volume data in memory, as the data is run-length-encoded for each axis independently: x, y, and z.

Raytracing methods use tree-like structures such as octrees, KD-trees and bounding volume hierarchies (BVHs) to compress the voxel data and accelerate the raytracing process. An octree-based raycaster proposed by A.Knoll at.al. [13] uses a pointer-based octree structure so that large iso-surfaces can be raycast interactively with high quality. The pointer-based octree structure is advantageous in that spatial queries can be made very efficiently. However, it needs to store at least one pointer (usually 4 bytes) for each node, which is more than two times the memory requirement of position data in typical RLE scenes.
As a variant of raytracing methods, Gigavoxels [11] uses bricks of volume data in combination with octrees to store voxels for interactive raytracing. The method suits well for raycasting large, semi-transparent volume data. Its features include filtering for high quality, and streaming on demand from the hard-drive to the GPU or CPU for enabling data sets larger than the CPU or GPU memory. However, it is not optimal for visualizing pure opaque surface data, because the used bricks store redundant transparent voxels as well, which increases the memory consumption.

One of the most well-known point based rendering methods, Qsplat[9], inspired many other researchers to propose similar rendering approaches. As an evolution of Qsplat, FarVoxels[10] improved the basic point-based-rendering by introducing a hybrid method that also utilizes polygonal rendering for geometries close to the viewpoint. However, as these methods employ either point-based rendering or a combination of point-based and polygonal-based rendering, they suffer from the earlier described disadvantages when compared with voxel-based rendering.

1.3 Proposed Approach

Since none of the related methods possess all of the following three properties: low memory consumption, high rendering performance (fast rendering) and optimal rendering quality, the purpose of our research is to find an optimal combination of all of the three technologies. Furthermore, we would like to improve the voxel data structure. We do not want to store the voxel's surface normal along with the voxel data. However, we still want to be able to recover the normal vector for lighting calculations. More specifically, our method should achieve the following goals:

(1) Highest quality of rendering voxel data by applying voxel smoothing and anti-aliasing.
(2) Significantly lower memory consumption than other methods, which are able to achieve highest quality, such as raycasting.
(3) High rendering performance even in complex environments at interactive frame-rates from arbitrary viewpoints.
(4) Support for recovering the voxels' surface normals from the depth buffer.

Our proposed approach is based on the so-called "voxel-based forward projection algorithm" developed by Wright et al. [1], which renders voxel data with lower memory consumption than the Shear-Warp algorithm. We further modified the original voxel based forward projection algorithm to deal with completely arbitrary voxel data, as is done in the unpublished work of Ken Silverman [2]. The original forward projection algorithm categorized the data into two groups: terrain, and objects that were placed on the terrain, such as trees and buildings. Each of these two groups of data had its own rendering technique. In [2] and in our approach, voxel data is stored in a uniform way as RLE data. An advantage of storing the data in a uniform way as opposed to categorizing the data into groups is, the data can be rendered using the same algorithm, hence reducing implementation complexity.

In this paper, we intend to achieve our goals by applying the following novel improvements to the voxel-based forward projection algorithm:

- Integrating the entire algorithm on the GPU
- Adding two novel culling algorithms to prevent unnecessary processing of occluded RLE elements
- Adding a novel smoothing filter for the removal of block-like artifacts of voxels close to the screen
- Include the recovery of surface normals from the depth buffer as a rendering post process

To accelerate the rendering process, our approach, for the first time, integrates the entire rendering algorithm on the GPU by using NVidia's CUDA[6] and the Pixel Shader. Until recently, graphics hardware was incapable of supporting random writes, which are crucial for the proposed method.
To improve the speed, we integrated three novel algorithms for the purpose of skipping occluded RLE

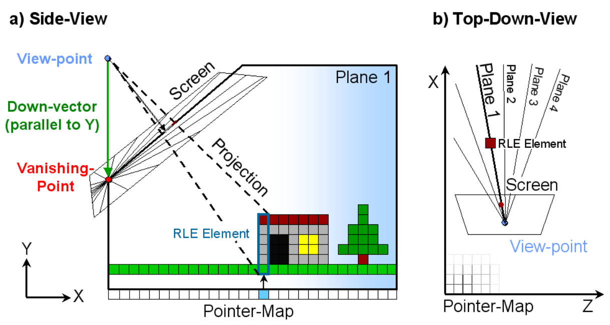

Fig.1: We raycast the scene in planes perpendicular to the x-z-plane. Each plane maps to the screen as a single line.

elements during rendering.
In order to remove the blocky appearance of voxels near the camera, we implemented a novel voxel smoothing method that is performed on the GPU as a post-process. Surface normals for light calculations are normally saved in the voxel data structure. However, to save memory, our algorithm does not store the surface normals, but recovers them from the scene's depth-buffer in real-time as a post-process. It is the first time that this post-process has been successfully implemented in real-time. Also, this is the first time that this process has been implemented on the GPU

1.4 Organization

The paper is organized as follows. Section 2 outlines the proposed method. Section 3 explains the pre-processing. Section 4 elaborates on the rendering by the GPU, Section 5 evaluates the proposed method experimentally, and Section 6 concludes this paper.

2. Overview

As shown in Fig. 1, the 3D surface voxel data exists in the x-y-z world coordinate system, where the x-z plane is the horizontal ground plane.
As shown in Fig.2, the algorithm consists of a series of steps, starting with the pre-processing step and ending with rendering the scene and changing the viewpoint. In the pre-processing step, the voxel data is run-length-encoded for each LOD in the vertical (y) direction. It is important that the encoding direction is vertical, because this leads to a higher average speed of the algorithm for the general case, when the camera looks towards the horizon. The details of this pre-processing step are described in Section 3.
As shown in Fig. 2, after copying the RLE data to the GPU's memory, the loop for visualizing the RLE data from the viewpoint at each time instant starts. As can be seen in Fig. 1, our proposed method visualizes the scene in planes that are perpendicular to the x-z plane and share the straight line that passes through the viewpoint

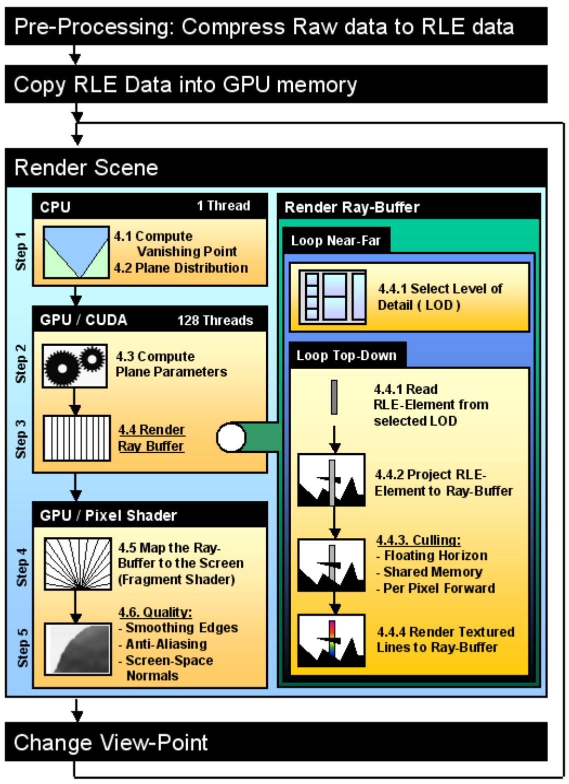
Pre-Processing: Compress Raw data to RLE data
Copy RLE Data into GPU memory
Render Scene
CPU
1 Thread
Step 1
4.1 Compute Vanishing Point
4.2 Plane Distribution
GPU / CUDA
128 Threads
Step 2
4.3 Compute Plane Parameters
Step 3
4.4 Render Ray Buffer
GPU / Pixel Shader
Step 4
4.5 Map the Ray-Buffer to the Screen (Fragment Shader)
Step 5
4.6. Quality:
- Smoothing Edges
- Anti-Aliasing
- Screen-Space Normals
Render Ray-Buffer
Loop Near-Far
4.4.1 Select Level of Detail (LOD)
Loop Top-Down
4.4.1 Read RLE-Element from selected LOD
4.4.2 Project RLE-Element to Ray-Buffer
4.4.3. Culling:
- Floating Horizon
- Shared Memory
- Per Pixel Forward
4.4.4 Render Textured Lines to Ray-Buffer
∞
Change View-Point

Fig.2 Pipeline: Everything except plane parameters is computed by the GPU. The numbers used in the figure link to the corresponding sections in this paper.

and is parallel to the y-axis (*Down-vector*). Raycasting the RLE data in each concentric plane is done step-by-step from near to far along the x-z plane, while the rasterization is done for each step in the vertical direction (parallel to the y axis) from top to bottom. To be more specific, for each step in the x-z-plane, all the RLE elements in the corresponding column are rasterized by projecting them into the screen space. Since the projection of each concentric plane is a line slanted across the screen space, the results of rendering the planes are stored as temporary bitmap for performance reasons. The temporary bitmap is then mapped to the screen using the Pixel Shader.
The render loop consists of the following five pipe-lined major steps referenced as 4.1 to 4.6 in Fig. 2. *Note that the corresponding sections and subsections are indicated in the parentheses.*

Step 1. Compute the vanishing point of all concentric plane's around the down vector on the CPU. As shown in Fig. 1, the vanishing point *vp* is the intersection of the screen plane and Down-vector. The vanishing point needs to be computed first (Section 4.1), before the concentric planes are computed (Section 4.2).

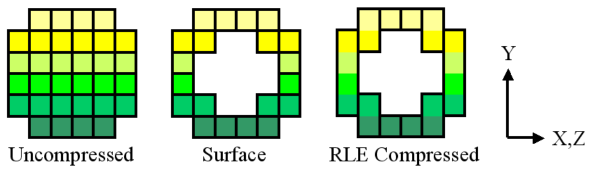

Fig.3: Pre-Processing: The initial volume data (left), then the surface in the center and the run-length-encoding of opaque segments on the right

Step 2. Compute the concentric plane parameters on the GPU: Each concentric plane's parameters, which are needed for the rendering process, are computed on the GPU (Section 4.3).

Step 3. Render the planes on the GPU: In each concentric plane, a ray is cast in the x-z plane from the viewpoint's x-z-coordinates to the maximal view-distance. For each x-z-position, the corresponding column of all RLE elements is rasterized from top to bottom for the selected LOD (Section 4.4.1) at this distance. For each RLE element, we first perform the projection of the coordinates to the ray-buffer (Section 4.4.2), then, culling is performed (Section 4.4.3). Finally, the element is rasterized as a textured line in the ray-buffer (a temporary bitmap) (Section 4.4.4).

Step 4. Display the temporary bitmap on the screen: The GPU-Pixel Shader is used to rearrange the rows of the temporary texture to a radial pattern of straight lines centered at the vanishing point on the screen (Section 4.5).

Step 5. Improving quality:
In the post-processing step, smoothing of voxels is performed to reduce their blocky appearance while anti-aliasing is included to further improve the rendering quality (Section 4.6).

In order to allow light-calculation without storing normal vectors inside the RLE volume data, we included a special method to recover the normal vectors from the depth buffer (Section 4.6.3).

3. Pre-Processing

3.1 Organization

The original source data to be visualized can either be volume data or polygon data. In case of polygonal data, the voxelization is simply done by rasterizing each triangle as voxels into volume data. As described earlier, we need to compress the voxelized data in the vertical (y-axis) direction from top (large y coordinates) to bottom (small y coordinates) using run-length encoding (RLE). More specifically, each vertical RLE

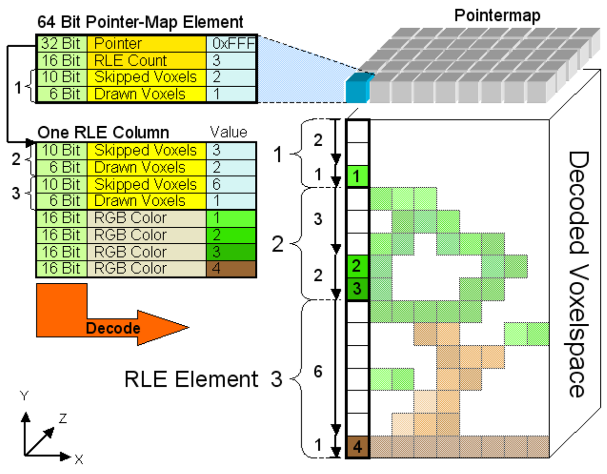

Fig.4: Data structure: The RLE data structure consists of a 64 Bit pointer-map referring to the RLE columns. Each RLE column consists of the RLE elements array and the depending color array.

column is compressed separately and referenced by one pointer of a two-dimensional lattice placed in the x-z plane, where the scale-factor of the lattice for the x and z directions are normally uniform, respectively. As shown in Fig. 3, not all the voxels of a solid volumetric object is run-length-encoded. To avoid unnecessary memory consumption, only voxel surface data is RLEed, while occluded voxels in the inner area are removed, i.e., not RLEed. Section 3.2 elaborates on the specific data structure of the RLEed voxel data.

3.2 Data Structure

The data structure of the voxel data should be able to utilize GPU's performance as much as possible, and it has therefore been optimized based on statistical evaluations of experimental results.
As shown in Fig. 4, the entire data structure basically consists of two parts: the pointer map and the RLE columns. Each element of the pointer map (the lattice in the x-z plane) stores three different variables: the pointer to its corresponding RLE column buffer (described below) the number of RLE elements (defined below) included in that RLE column as well as the first (top-most) RLE element, consisting of "skipped voxels" and "drawn voxels". This paper defines an RLE element as a series of sequential skipped voxels and sequential drawn voxels, where a skipped voxel corresponds to an invisible voxel that is not stored, and a drawn voxel corresponds to a voxel that is stored in the RLE structure with RGB color data. For example, in the decoded voxel-space illustrated in the right side in Fig. 4, white voxels indicate skipped voxels and colored voxels indicate drawn voxels, respectively. In the left-most voxel column, the two voxels from the top are skipped (not drawn), and just below there is one

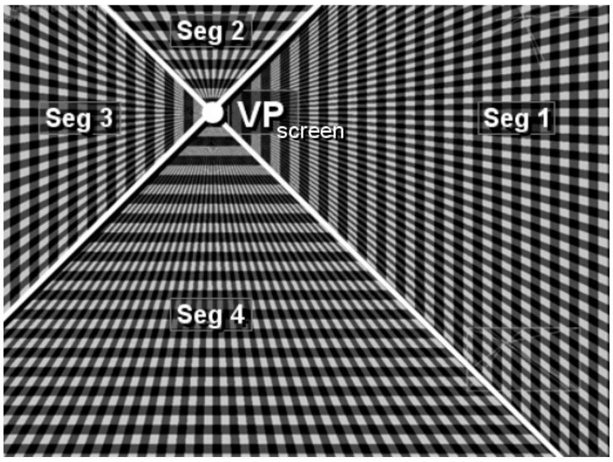

Fig.5 Screen segmentation: VP represents the vanishing point while Seg 1..4 refer to the four segments.

(colored) drawn voxel. Therefore, “2” and “1” are stored in the “skipped voxels” field and “drawn voxels” field in the pointer map element of the RLE structure in the left side, respectively.
As shown in Fig. 4, the RLE column’s buffer, which is referenced by the pointer of an element of the pointer map, stores the numbers of the skipped voxels and drawn voxels of the second to the *m*-th RLE elements, where *m* is equal to the number of the RLE elements stored in one column, the element of the pointer map. In addition, the RLE column’s buffer stores the color for each drawn voxel in the order of the voxels’ appearance in the RLE column.
To achieve efficient computation by GPU, the number of memory accesses has to be minimized. We therefore store 64 bit elements in the pointer-map, as 64 bit is the largest amount of memory that can be pulled in one read by the GPU. Note that one 64-bit element includes all the data required to test the visibility of the first (topmost) RLE element. This strategy increases the rendering performance (speed) particularly for large outdoor environments and landscape-like scenes with hills and mountains, because one memory read is sufficient to test the visibility for approximately 90% of all rasterized elements according to our preliminary studies.

3.3 Level-of-Detail Computation

As described in Sections 1.3 and 2, the individual RLE data for each level of detail is obtained in advance prior to the visualization process. We apply the idea of texture mip-maps to the original RLEed voxel data and generate RLEed mip-volumes. The original RLEed voxel data has the highest resolution and is used for the LOD that corresponds to the range closest to the view point. As the distance from the viewpoint gets larger, RLEed voxel data with lower resolutions are used. More specifically, suppose that lev denotes a level of detail, where lev ranges from 1 (highest resolution) to L (lowest resolution); the size (length of a side) of one voxel in the level lev (≧2) is twice as large (long) as that in the level lev-1, where linear down-sampling is applied to the voxel data in the level lev-1 so that the voxel data in the level lev is obtained. For example, an original volume of 16x16x16 has four mip-volumes: 8x8x8, 4x4x4, 2x2x2 and 1x1x1. As described in the following, the resolution is dynamically chosen by the visualization process, depending on the distance to the viewpoint.

4. Rendering

The rendering for each frame consists of multiple steps, as displayed in Fig.2 and described in Section 2.

4.1 Vanishing-Point

We first compute the vanishing point vp, the point at which all the concentric planes meet in the screen plane (see Fig.1). Each plane is projected to the screen as one straight line and all the lines meet at the vp. The vanishing point can easily be obtained by intersecting the vertical line that is parallel to the y-axis and passes through the viewpoint with the screen-plane as follows.

$$vp = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \cdot \frac{-d}{\sin(\alpha_p)} \tag{1}$$

where d denotes the distance between the camera origin (view point) and screen-plane, and αp represents the camera's pitch angle, which is defined as the rotation around the horizontal axis (the x-axis) of the camera coordinate system. A pitch angle of zero means that the optical axis of the camera is horizontal. The vanishing point is projected to the screen space by the following equation:

$$vp_{screen} = A_{cam} \cdot vp \tag{2}$$

where vp_{screen} represents the projection of vp to the screen space, A_{cam} represents the 4x4 camera matrix. Each plane intersects the screen as one line originated in vp_{screen} (Fig.1).

4.2 Concentric Planes

Since each plane is projected to the screen as one line that is originated in vp_{screen}, we need to focus on achieving a complete coverage of the screen by the lines originated in vp_{screen}. To achieve this, as shown in Fig. 5, we split the screen into four segments, where the borderlines between adjacent segments meet at vp_{screen}, and the angle between adjacent borders is 90 degrees. We texture each line included in the left and right (with respect to vp_{screen}) segments in the horizontal direction and in the upper and lower segments in the vertical direction. The number of lines included in each segment depends on the number of pixels on the screen border in this particular segment. This implies that each pixel in the screen border of a segment should be the end of a line (projected plane), whose another end is vp_{screen}. We can calculate the number of planes (lines) as indicated below:

$$np_i = 2 \cdot dist(vp_{screen}, border_i), i \in [1..4] \qquad (3)$$

where np_i denotes the number of planes for a given $segment_i$, $border_i$ denotes one of the four borders of the screen, and dist(…) indicates the computation of the distance in pixels between vp_{screen} and $border_i$. The parameters vp_{screen} and np_i, which are computed by CPU, are transferred to the GPU for the subsequent computations.

4.3 Plane Parameters

As described in Section 2, all the calculations described in the rest of Section 4 are executed on the GPU in a parallel manner by using multiple threads. The number of simultaneous running threads depends on the number of processing units of the underlying hardware. In our case, 240 processing units are available.

The parameters to be computed for each plane are as follows: the start and end points of the projected line in the screen as well as the plane's rotation around the y-axis. The start and end points are used for rendering and clipping the projected RLE elements to the screen. The rotation around the y-axis defines the orientation in which we march through the RLE structure (Section 4.4).

4.4 Rasterizing the Ray Buffer

The RLE elements are visualized in two steps. In a first step we rasterize the elements to a 2D temporary ray-buffer, each row of which stores the projected result of one concentric plane. In a second step, the temporary ray-buffer's contents are texture-mapped to the screen.

4.4.1 Traversal per Plane

To rasterize the RLE elements to the temporary ray-buffer, we traverse the pointer-map, which is placed in the x-z plane as shown in Fig. 1 and Fig. 4. As shown in Fig. 1, the straight line in which a concentric plane and the pointer-map (x-z plane) meet is considered. For

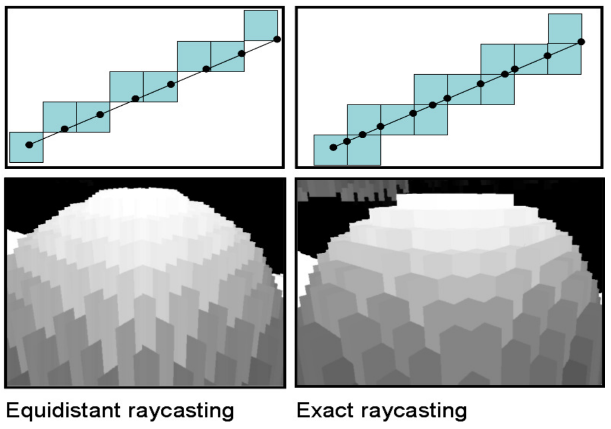

Fig.6: Ray sampling: On the left side we can see the simple equidistant raycasting while the more advanced and accurate one is shown on the right.

a point (an element of the pointer-map) on the straight line, the RLE elements (voxels) visible from the viewpoint are rasterized in the radial line in which the concentric plane and the screen meet. This process starts from the point just below the viewpoint and is traversed till it reaches the point that corresponds to the predefined maximal distance from the viewpoint. During this traversal, culling, which is explained in Section 5, is performed for the visibility check.

The traversal is not equidistant as it is often done in volume visualization. As shown in Fig. 6, equidistant traversal performs equidistant sampling of the pointer-map's elements on the straight line. This is simple, but leads to errors in the visualization. Instead, we apply an exact grid traversal, which correctly samples all the 2D grid intersections during the traversal according to [4]. In Fig.6 we compare the visual results of the exact traversal and the equidistant traversal. The exact traversal requires slightly more computational effort, but the result is significantly better.

During the above-mentioned traversal, LOD needs to be switched according to the distance from the viewpoint. This paper selects LOD according to the distance between the viewpoint and a point on the line in which the concentric plane and the x-z plane meet. Suppose *pd* is a predefine distance along the line. From P0, the point below the view point, to P1, which is away from P0 by *pd* on the line, the RLE data (voxels) with the highest resolution is used for the rasterization; similarly, from P1 to P2, which is away from P1 by *pd*, the second highest resolution is used, etc.

4.4.2 Projecting RLE Elements to Ray-Buffer

As mentioned earlier, the visible part of each RLE element is rasterized to the temporary buffer as a textured line, where the x, y and z coordinates of the start-point *ps* and the end-point *pe* are the 3D world

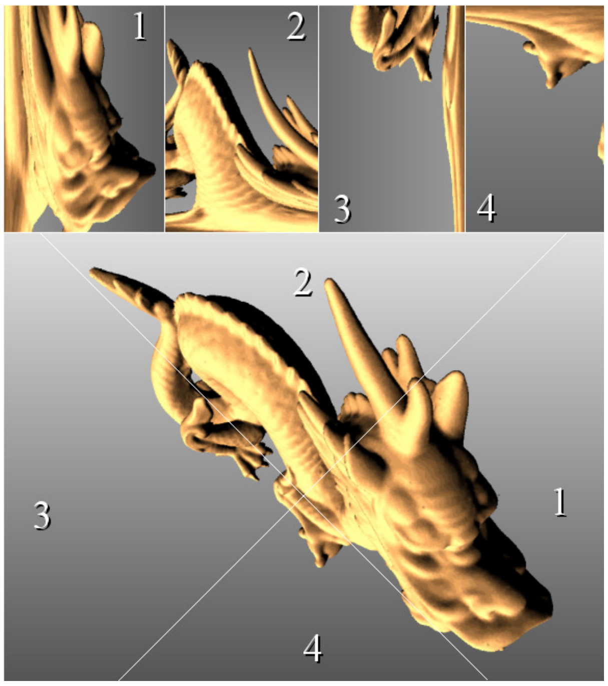

Fig.7: Ray mapping: The upper part shows the temporary buffer with the four segments. The lower half demonstrates how the segments are mapped to the screen.

space coordinates of the particular RLE element. We project *ps* and *pe* into the screen-space using the camera matrix A_{cam} as follows:

$$\begin{aligned} ps_{cam} &= A_{cam} \cdot ps \\ pe_{cam} &= A_{cam} \cdot pe \\ ps_{screen} &= \begin{bmatrix} ps_{cam}.x \\ ps_{cam}.y \end{bmatrix} \cdot \frac{1}{ps_{cam}.z} \\ pe_{screen} &= \begin{bmatrix} pe_{cam}.x \\ pe_{cam}.y \end{bmatrix} \cdot \frac{1}{pe_{cam}.z} \end{aligned} \tag{4}$$

In the formula, ps_{cam} and pe_{cam} contain the x, y and z coordinates of the *ps* and *pe* in the camera space. The camera space is defined as orthonormal-basis, where the origin is placed at the view-point, the z-axis a straight line from the viewpoint towards the center of the screen, the x-axis a straight line towards the origin and parallel to the upper and lower screen border and the y-axis a straight line towards the origin and parallel to the left and right screen border. The variables ps_{screen} and pe_{screen} are the two dimensional ray-buffer coordinates of *ps* and *pe*. As described in Section 4.2, either the horizontal (x) or vertical (y) component of the start and end coordinates is used for rasterizing RLE elements into the ray-buffer. In the ray-buffer, the projection of each plane is represented as one column, as shown in the upper half of Fig.7. Therefore, either the horizontal (x) or vertical (y) coordinates of the start and end-point are used to define the vertical 1D position inside the column of the ray-buffer. In Fig.7, Segments 1 and 3 use the horizontal (x) coordinate, while Segment 2 and 4 use the vertical (y) coordinate of ps_{screen} and pe_{screen}. After the start and end positions inside the column are determined, visibility culling is performed (detailed in Section 4.4.3), before the textured rasterization is done (Section 4.4.4).

4.4.3 Culling

As described in Section 4, culling needs to be performed to only render the visible parts of RLE elements and efficiently skip RLE elements that are invisible. This paper uses three culling methods including novel and known methods. It is possible to combine these culling methods for optimal performance. However, utilizing all the algorithms simultaneously is not efficient due to mutual interference. It is efficient to use the floating horizon algorithm together with shared memory culling or per pixel forwarding. However, shared memory culling and per pixel forwarding interfere, because they are both executed on a per-pixel-level.

4.4.3.1 Modified Floating Horizon

We utilize the well-known floating horizon algorithm, which has already been used in the original voxel forward projection algorithm [1]. The floating horizon algorithm does not conflict with the other culling methods we use and can hence be used in combination with all the other culling methods. The algorithm works as follows:

For each rendered plane, we store two offset values: one start and one end-offset along the projected line in the screen, defining the bounds of the render-able area. Once one RLE element that touches the start or end offset is drawn, we update this particular offset to narrow the bounding area along the line, which allows to cull more RLE elements. Using the floating horizon algorithm is possible, because we render opaque scenes from near to far, which means that every pixel is drawn only once.

However, the basic floating horizon algorithm only works well if we have a height-map like scene such as mountains. In case of complex scenes such as a tree, we have the problem of unconnected segments rasterized along the line, which cannot be handled efficiently by the original algorithm. We therefore introduce a small but significant modification to the original method so that good performance is achieved even in complex scenes. The modification is as follows: after one RLE element is rasterized that touches either border, we update the

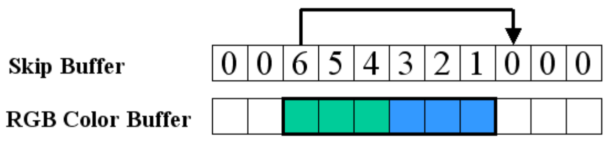

Fig.8: Skip-Buffer: Each element in the skip-buffer stores the number of drawn pixels that can be skipped until a free pixel is found.

offsets and further test pixels next to the new offsets if they have been drawn already. If they have been drawn already, we further narrow the bounds. Depending on the scene, this modification accelerates the culling process up to two times.

4.4.3.2 Shared Memory

The shared-memory culling algorithm takes advantage of the fact that our method draws every pixel in the screen only once. This means a binary map suffices to store the visibility information in the screen. This map is so small that it is fitted entirely into the graphic cards shared memory. Our target hardware, the NVidia GTX series, provides two main types of memory: Global memory and shared memory. The difference between both types is that a memory access to global memory consumes about 300 processor cycles, while an access to the shared memory only requires one cycle. Therefore, using a binary visibility map stored in the shared memory, we can apply per pixel culling very fast without accessing the slower main memory. Shared memory culling accelerates the rendering speed by 40% to 140%, depending on the scene.

4.4.3.3 Per Pixel Forward

Lacroute's culling based on per pixel forwarding [3] is slightly slower and more complex than the previously described shared memory culling, but it is needed for screen-resolutions where the number of simultaneously processed pixels of the screen exceeds the number of bits available in the shared memory. In our case, this occurs at screen resolutions with more than 900 pixels in the vertical direction.

The per-pixel forward algorithm works as follows: for each pixel in the temporary buffer we store a relative jump offset. This offset is set to zero in the beginning and is updated once an RLE element is drawn as shown in Fig.8. Since relative jumps help to skip pixels efficiently, we achieve a speed-up of approximately 1.08 to 2.0 times, which is significantly faster than the floating horizon algorithm alone, but approximately 20% slower compared to shared-memory culling.

4.4.4 Drawing RLE elements as textured Lines

Each RLE element is rasterized into one or multiple columns of the temporary ray buffer as a texture mapped line, using the coordinates of ps and pe as the vertical positions in the column. Using texture mapping significantly speeds up the overall computation, because voxels are rendered as a group rather than individually (the data structure is described in Section 3.2). To achieve a proper appearance, we apply perspective correct texture mapping. Simple non-perspective texture mapping interpolates the 2D texture coordinates, which leads to an approximated but wrong visual appearance. Perspective correct texture mapping uses not only the 2D texture coordinates but also the depth coordinate (z), which leads to a correct result.

4.5 Displaying the Ray-Buffer

We can map the texture stored in the temporary ray buffer efficiently to the screen using the graphics card's Pixel-Shader. To achieve this, we have to calculate the source *(U,V)* texture coordinate in the ray buffer for each target pixel *(xs,ys)* on the screen. The mapping is applied in a concentric manner with respect to the vanishing point vp as shown in Fig.7. We define the formula to compute the source (U,V) texture coordinates inside the ray-buffer by Eq.(5) as follows.

$$\begin{aligned} U_{2,4} &= (\mathrm{xs}-\mathrm{vp.x})\cdot|\mathrm{ys}-\mathrm{vp.y}|+s_{2,4} \\ V_{2,4} &= \mathrm{ys}-\mathrm{vp.y} \\ U_{1,3} &= (\mathrm{ys}-\mathrm{vp.}y)\cdot|\mathrm{xs}-\mathrm{vp.}x|+s_{1,3} \\ V_{1,3} &= x\mathrm{s}-\mathrm{vp.}x \end{aligned} \tag{5}$$

In the formula, *U* defines the horizontal coordinate inside the ray-buffer, *V* the vertical coordinate, *xs* the horizontal screen coordinate, *ys* the vertical screen coordinate and *s* the start-offset that is added for the corresponding segment of the ray-map. The indices of *U,V* and *s* represent the segment index as numbered in Fig 5. The valid range of the texture coordinates *(U,V)* as well as the screen coordinates *(xs,ys)* ranges from 0 to 1.

4.6 Quality Aspects

As shown in the flow-chart of Fig.2, the quality of the image rendered in the screen is improved at the final stage of the render pipeline. Since we use conventional texture mapping functions of the graphics card, texture filtering, which is natively supported by every GPU, can be applied without performance impact to improve the quality. We employ two methods to handle this issue: smoothing and anti-aliasing. The combination of both algorithms can significantly improve the rendered image quality.

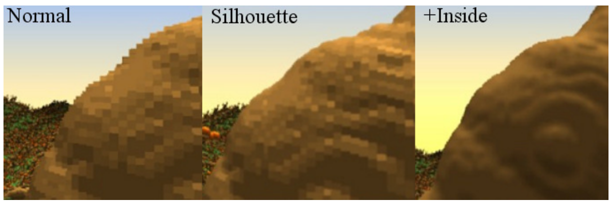

Fig.9: Smoothing: The left image shows the basic rendering. To improve the rendering quality, we smooth the silhouette (middle image) and the interior part as well (right).

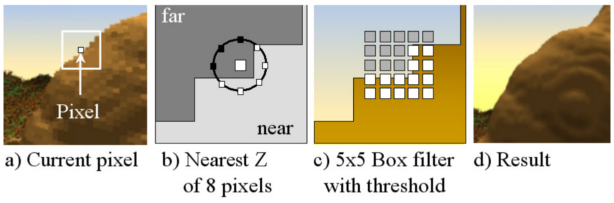

Fig.10: Smoothing steps: a.) Target pixel b.) Find minimum depth (Z) c.) Box-filter with threshold; scaled according to the minimum depth d.) Result

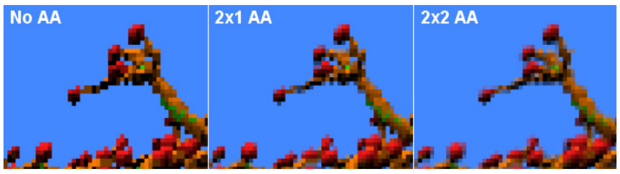

Fig.11: Anti-aliasing (AA): We investigated three quality levels: No AA (left), 2x1 AA (center) and 2x2 AA (right).

4.6.1 Smoothing

We apply smoothing as a post-process in image-space by the Pixel Shader. We developed a special smoothing method that achieves two types of smoothing in one shader pass: Smoothing of voxel silhouettes and smoothing of voxels close to the camera themselves. Figure 9 shows an example of the result of this method.

The smoothing contains of multiple steps, illustrated in Fig.10. Step a) shows the target pixel in the original image. In step b.), we search the minimum depth of eight pixels that lie in a circle around the target pixel. The radius is fixed for this operation. In step c) we apply a box filter for 5x5 pixels. The scale factor of the box filter is determined by the previously computed minimum-depth. For the smoothing, we only include pixels, which are within a limited depth range near to the minimum depth. As a result, we can see that both, the silhouette and the inner region in our example has been smoothed well in step d).

4.6.2 Anti-Aliasing

To further improve the quality, we apply full-screen anti-aliasing (AA) by rendering the scene with a higher resolution and down sampling the rendered image so as to obtain the target resolution. Figure 11 compares three configurations: No AA, 2x1 pixels AA and 2x2 pixels AA. Obviously, 2x2 pixels AA and 2x1 pixels AA give the best and second best quality.

4.6.3 Screen Space Normals (SSN)

To visualize large data sets such as the Richtmyer-Meshkov on consumer graphics cards with only 256MB ram, we can recover (approximate) the surface normal n for shading from a few samples in the depth buffer by Eq. (6).

$$\begin{aligned} z_s &= Depth(x_s, y_s) \\ \Delta x &= x_s - rnd(1/z_s) \\ \Delta y &= y_s - rnd(1/z_s) \\ dz_x &= z_s - Depth(x_s + \Delta x, y_s + \Delta y)/\Delta x \\ dz_y &= z_s - Depth(x_s + \Delta x, y_s + \Delta y)/\Delta y \\ n &= (1, 0, dz_x) \times (0, 1, dz_y) \end{aligned} \tag{6}$$

where x_s, and y_s represent the horizontal and vertical coordinates of a pixel in the screen, respectively, *Depth* (.) represents the depth of the pixel (argument) in the depth-buffer, *rnd* is the random function to achieve an averaged result for multiple samples, x for computing n is vector cross-product. Note that Eq. (6) indicates that the sample region needs to be reciprocal in size to the sampled depth value z_s of the pixel (x_s, and y_s). In case the pixel is close to the camera, we need a large region and vice versa.

To achieve a satisfying result in our implementation, at least 16 samples from the depth-buffer should be used. Since computing a random value by GPU is slow, we sample a random value from a texture as an alternative. As SSN and SSAO [12] sample the depth-buffer in a similar way, it is possible to efficiently combine both methods in only one shader-pass. An example of the result is demonstrated in Fig.12.

5. Experimental Results

We conducted experiments with multiple scenes to evaluate our algorithm in terms of rendering speed, memory consumption and quality. The scenes used for testing are shown in Fig.13. Our experimental system consists of a Pentium-D 3.0 GHz Processor with 1 GB of RAM and a GeForce285 GTX (1024MB) graphics board with 240 stream processors. As shown in Fig. 2, we use

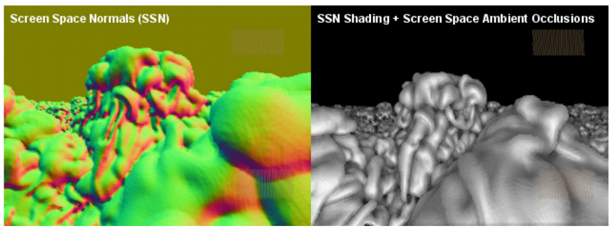
Screen Space Normals (SSN)
SSN Shading + Screen Space Ambient Occlusions

Fig.12: Normals: The depth-buffer can successfully be utilized to compute normal vectors on the fly (Left). These can be utilized for shading and further enhanced with screen-space-ambient-occlusions (Right).

NVidia CUDA to compute the raycasting part of the algorithm, while texture mapping the temporary ray-buffer and the post-processing are executed in the Pixel-Shader. The render-resolution for all the tests is set to 1024x768 pixels, while the AA setting for improving quality is 2x1, which provided the best tradeoff between quality and performance.

Table 1 shows the result of benchmark tests. Bits per voxel indicates the number of bits required for storing the position information of one voxel, taking the pointer-map and mip-maps into account as well. The bits used to store the position of one voxel range from 10.83 to 26.3, which is significantly less than a pointer-based octree. The pointer-based octree requires 32 bit only for the tree leaves, which sums up to about 32*(1+1/8+1/64+..)=36.8 bits for the entire tree. However, in some scenes our algorithm requires more memory than splatting based algorithms such as QSplat, which only utilizes 13 bits per leaf. As described earlier, the accuracy of splatting-based methods is limited to the size of the splats; therefore, in particular, unreasonably sharp edges tend to degrade the image quality.

To measure and evaluate the rendering speed, we first investigate the maximum polygon performance of our graphic card. In case of rendering as a quad by two textured triangles, 350 Million triangles per second are the limit of our graphic card for rendering triangle strips while splatting reaches 100 Million primitives (splats) per second. In Table 1 we can see that the proposed algorithm achieves a high count of processed RLE elements per second (Speed, Elems/s), ranging from 112 to 365.8 Million RLE elements per second. This speed is twice to three times as fast as basic splatting and surprisingly outperforms even the default OpenGL rendering pipeline with 350 Million triangles/s in certain cases. Further information included in the table are the total number of RLE elements inside the view frustum (RLE Elem total), the number of RLE Elements that have passed the culling test (RLE Elements, ren), frames per second (fps) and the resolution of the single dataset

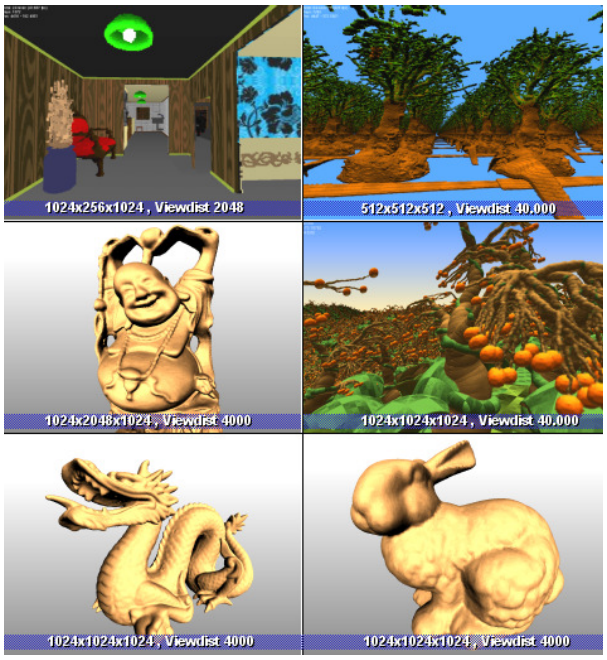
1024x256x1024 , Viewdist 2048
512x512x512 , Viewdist 40.000
1024x2048x1024 , Viewdist 4000
1024x1024x1024 , Viewdist 40.000
1024x1024x1024 , Viewdist 4000
1024x1024x1024 , Viewdist 4000

Fig.13: Test scenarios: Handcrafted mansion (up left), Bonsai forest with 3000 trees (upright), Happy Buddha (middle left) and a Procedural Landscape with about 4000 visible trees (middle right), the Stanford Dragon and the Stanford Bunny.

(Resolution). For testing the performance in case of large outdoor areas, we created scenes containing more than one thousand instances of the same data set for the procedural scene and the bonsai scene. The maximal view distance has been set to 40.000 voxel in both cases. The large outdoor scenes of the bonsai and the procedural dataset have contributed to table 1 as well. They have been used in all our tests, to evaluate the performance, the compression ratio and the quality as well.

To compare the performance of our GPU implementation to the CPU, we implemented our method also for the CPU as well. As a result, it turns out that the GPU implementation tested on an NVidia GeForce 285 is three to seven times as fast as the CPU implementation executed on a test system with an Intel Core2 Quad Q6600 CPU with 4x3 Ghz and 1GB ram. The GPU outperformed the CPU by factor three for light scenes without AA and factor seven for complex scenes, with AA enabled, where the scenes used for testing are shown in Fig.14.

We further evaluated the render speed in regard to the image quality by measuring the performance for different quality settings. We compared no anti-aliasing, 2x1 anti-aliasing and 2x2 anti aliasing (Fig. 11). If the speed for the no anti-aliasing is 100%, 2x1 AA and 2x2 AA achieve 104% and approximately 80%, respectively. The increase in speed for 2x1 AA might be explained as

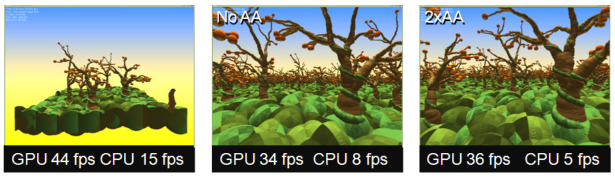

Fig.14: GPU vs CPU: We compared our GPU implementation running on an NVidia GTX 285 to the CPU (Intel Q6600 4x3Ghz).

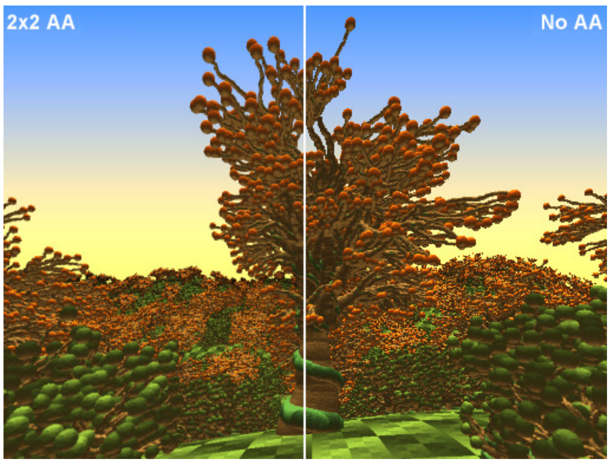

Fig.15: Quality: To show the ability to render at high quality, we created a complex test scene with many fine details rendered at 512x384 pixel and 2x2 AA as well as no AA for a comparison. Note that 2x2 AA successfully removes aliasing artifacts for distant pixels.

Scene	RLE Elements (in Mill.)			Fps	Speed	Resolution	Compression
1024x768, 2xAA	Total	Proc	Ren		Elems/s	x/y/z	Bit/Voxel
Procedural	14.1	7.7	0.58	47.5	365.75	1k/1k/1k	17.9
Bonsai	8.8	8.7	0.35	21.7	188.79	512/512/512	18.25
Budda	31.3	7.38	0.4	48.2	355.72	1k/2k/1k	12
Mansion	11.48	3.1	0.43	78.5	243.35	1k/256/1k	10.83
Dragon	2.67	1.67	0.29	67.1	112.06	1k/1k/1k	26.3

Table 1: Performance: We evaluated the performance of our algorithm based on various scenes. Anti-aliasing is set to 2x1.

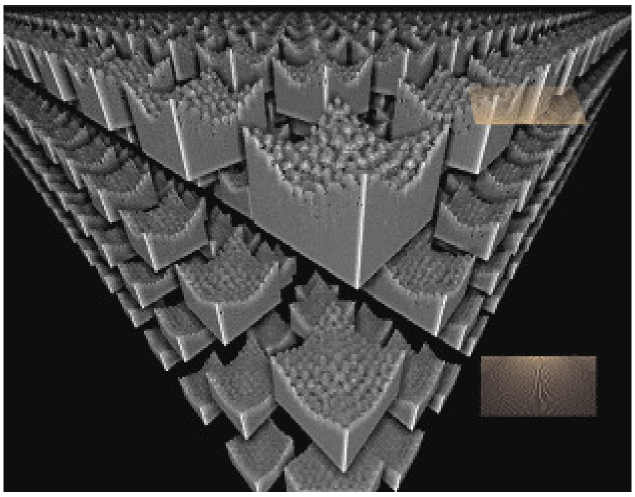

Fig.16: It is possible to render the complete Richtmyer-Meshkov dataset more than 100 times at interactive rates at a 1024x768 screen resolution. For the shading, we utilized a combination of screen-space-ambient-occlusion and screen-space normals.

better coalescence for reads from GPU memory. On the GPU, coalescent memory reads are very important. It also means that half the GPU's processing units are idle in case of the no AA configuration, because 2x1 AA requires two times as many floating-point operations as no AA. As a result of this experiment, the main limiting factor of our algorithm is the memory-bandwidth. In computer graphics, every rendering algorithm's speed is either limited by the speed of the processing unit (here the GPU) or the speed of the memory. In out case, the speed of the memory is the limitation. We already reduced the memory bandwidth by employing multiple culling algorithms – yet it still remains the limiting factor. To further improve on that, additional compression schemes might be helpful.

As shown in Fig. 15, our algorithm is able to achieve high quality renderings for a scene with many fine structures. To facilitate the comparison, we render the result using 2x2 AA in the left half and no AA in the right half.

Finally we visualized the Richtyer-Meshkov data set, which is one of the largest data sets, with a resolution of 2048x1920x2048. The size of the RLE compressed data of the surface at iso-value 60 is 198 MB including mip-maps. This results in a compression factor of 5:1 in regard to the binary volume data. As this particular data set is very large, we do not store color or shading information. For the visualization we compute the normal vectors on the fly from the screen-space, as well as approximated ambient occlusions. The visualization speed that we achieved at a resolution of 1024x768 is interactive frame-rates: 15 fps for rendering a single instance of the data-set (Fig.14) and 10 fps for rendering the data-set repeatedly as shown in Fig. 16.

6. Conclusions

This paper presents an efficient adaptation of the voxel forward projection algorithm to utilize recent graphics hardware. Several improvements to speed up the algorithm, optimize memory consumption, as well as improve the rendering quality were proposed. Initially, the surface data is run-length-encoded to allow fast decoding for rendering on the GPU. According to [8], RLE is the second fastest algorithm to decode lossless compressed volume data.

Experiments using different scenes show the following results. The proposed algorithm is up to seven times as fast as the equivalent CPU implementation, up to three times as fast as basic splatting and even surpasses the performance of the default OpenGL pipeline in terms of primitives per second for some of our test scenes. We were able to further visualize large and highly detailed data sets on a single NVidia GTX 285 GPU at interactive frame-rates.

Furthermore, our method is very memory efficient: i.e., the storage required for one voxel during run-time is only 10.8 to 26 bits (Table 1). This is on average slightly more than Qsplat, which achieves 13 bits, but has significantly higher quality (Fig.9 and Fig.15). Also, the storage per voxel is significantly less than conventional tree-based ray casting (such as octree), which consumes more than 32 bits for referencing each node.

In terms of quality, we achieve sub-pixel accurate rendering by employing 2x1 full-screen anti-aliasing at full speed. Using 2x1 AA does not decrease the speed compared to not using AA.

Although the results are very promising, remaining issues include adding support for streaming data into the GPU memory on demand. This allows for rendering scenes that do not fit inside the graphic card's memory.

Acknowledgements

We would like to thank Ken Silverman for his supporting help during the initial phase of this research, Mark Duchaineau for providing us the Richmyer-Meshkov instability data-set for testing, NVidia for sponsoring our project with graphics-hardware and finally the Honjo International Scholarship Foundation for making the whole research possible.

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, visited Feb.2010

[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/, visited Feb.2010

[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 and Eisemann, Elmar:"GigaVoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering", ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D), Feb.2009, to appear

[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

[14] T.Todd Elvins:"A survey of algorithms for volume visualization", ACM SIGGRAPH 1992, Vol.26 , Issue 3, pp. 194 - 201, 1992 , ISSN:0097-8930

[15] The Megatexture technology, interview with John Carmack, May 1st, 2006. Link visited on 2010/01/27: http://www.team5150.com/~andrew/carmack/johnc_interview_2006_MegaTexture_QandA.html

Sven Forstmann. Sven Forstmann was born in Konstanz, Germany on Jan. 26th 1977. He was awarded an M.S. degree in computer science from Karlsruhe University in the year 2004. From 2004 to 2005 he was a special research student at the GITS faculty of the Waseda University in Tokyo, before he began a Ph.D course at the GITS faculty in 2005. His specializations are in the areas of Computer Graphics and Image Processing.

Ohya, Jun: Jun Ohya received B.S., M.S. and Ph.D. degrees in precision machinery engineering from the University of Tokyo, Japan, in 1977, 1979 and 1988, respectively. He joined NTT Research Laboratories in 1979. He was a visiting research associate at the University of Maryland, USA, from 1988 to 1989. He transferred to ATR, Kyoto, Japan, in 1992. Since 2000, he has been a professor at Waseda University, Japan. He was a visiting professor at the University of Karlsruhe, Germany, in 2005. His research areas include computer vision, computer graphics and virtual reality. Dr. Ohya is a member of IEEE, IPSJ and VRSJ.