PAPER

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

Sven FORSTMANN^{†a)}, *Nonmember* 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 compressed using run-length encoding (RLE) for each level of detail (LOD). Then, the loop for the rendering process is performed on the GPU for the position of the viewpoint at each time instant. The scene is raycasted in planes, where each plane is perpendicular to the horizontal plane in the world coordinate system 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 up to the maximum view distance. This rasterization process projects each RLE element passing the occlusion test onto the screen at a LOD that decreases with the distance of the RLE element from the viewpoint. Finally, the smoothing of voxels in screen space and full screen anti-aliasing is 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 have shown that the proposed algorithm is faster than the equivalent CPU implementation and other related methods. Our experiments further prove that this method is 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 or rendering speed. This means that voxel- and point-based rendering methods gain more importance 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.

One advantage of polygon-based rendering is the ability to use repeating textures to save memory. Voxel- and splatting-based rendering inherently use unique texturing,

so there is no benefit in memory consumption from using repetitive texturing. However, the recent trend of using unique, non-repeating textures for each object on the screen (Megatexture technology [1]) has sharply increased the memory consumption for polygon-based rendering. 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 LOD between voxels and polygons. LOD is important for accelerating the rendering and decreasing the run-time memory consumption. For polygonal objects, LOD is usually implemented as follows. First, a set of polygonal objects with different LODs 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 with lower details. Therefore, it is not necessary for an artist to design separate models of the same object for each LOD; the different LODs can be generated automatically.

The final advantage of voxels over polygons is that Boolean operations can be applied much more easily.

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 is that voxels occupy a well-defined cubic portion of volume in space, whereas point-based methods approximate the geometry, usually by using 2D splats.

Since splats are 2D approximations of a 3D object, several exceptions must be 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 voxel volume data by using Shear-Warp [2], raytracing-based algorithms and point-based rendering. We will briefly review the most widely used methods in each group and mention their key issues. Shear-Warp [2] renders RLE volume data in a front-to-back manner to a temporary texture.

Manuscript received February 18, 2010.

Manuscript revised June 18, 2010.

[†]The authors are with the Graduate School of Global Information and Telecommunication Studies, Waseda University, Tokyo, 169-0051 Japan.

a) E-mail: sven@ddd.co.jp

DOI: 10.1587/transinf.E93.D.3088

This temporary texture is then mapped to the screen. This 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, because the data are RLE compressed for each axis independently: X, Y, and Z.

Raytracing methods use tree-like structures such as octrees, KD-trees, and bounding volume hierarchies to compress the voxel data and accelerate the raytracing process. An octree-based raycaster proposed by Knoll et al. [3] uses a pointer-based octree structure so that large iso-surfaces can be raycast interactively with high quality. This pointer-based octree structure is advantageous in that spatial queries can be processed very efficiently. However, it needs to store at least one pointer (usually 4 bytes) for each node, which is more than twice the memory requirement of position data in typical RLE scenes.

As a variant of raytracing methods, Gigavoxels [4] use bricks of volume data in combination with octrees to store voxels for interactive raytracing. This method works 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 datasets larger than the CPU or GPU memory. However, it is not optimal for visualizing pure opaque surface data because the used bricks store also redundant transparent voxels, which increases the memory consumption.

One of the best-known point-based rendering methods, Qsplat [5], inspired many other researchers to propose similar rendering approaches. As an evolution of Qsplat, Far-Voxels [6] improved basic point-based rendering by introducing a hybrid method that also uses polygonal rendering for geometries close to the viewpoint. However, because these methods employ either point-based rendering or a combination of point-based and polygonal-based rendering, they suffer from the earlier described disadvantages compared with voxel-based rendering.

1.3 Proposed Approach

Since none of the related methods demonstrate low memory consumption, high rendering performance (fast rendering), and optimal rendering quality, the purpose of our research is to find the optimal combination of all 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. To reach the highest quality of rendering voxel data by applying voxel smoothing and anti-aliasing.
2. To significantly lower memory consumption compared with other methods that are able to achieve highest quality, such as raycasting.
3. To complete high rendering performance even in complex environments at interactive frame rates from arbitrary viewpoints.

4. To support the recovery of 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. [7], 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, similar to the unpublished work of Ken Silverman [8]. 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 [8] and in our approach, voxel data are stored in a uniform way as RLE data. One advantage of storing the data in a uniform way as opposed to categorizing the data into groups is that the data can be rendered using the same algorithm, thereby 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 most important parts of the algorithm on the GPU;
- Adding two novel culling algorithms to prevent the unnecessary processing of occluded RLE elements; and
- Including the recovery of surface normals from the depth buffer as a rendering post-process.

To accelerate the rendering process, our approach integrates the majority of the rendering algorithm on the GPU by using NVidia's CUDA [9] and the Pixel Shader. Until recently, this graphics hardware was incapable of supporting random writes, which are crucial for the proposed method. To improve the speed, we integrated three novel algorithms to skip occluded RLE elements during rendering. To remove the blocky appearance of the voxels near the camera, we implemented a special 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.

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 Sect. 6 concludes.

2. Overview

As shown in Fig. 1, the 3D surface voxel data exist in the X-Y-Z world coordinate system, where the X-Z plane is the horizontal ground plane.

Fig. 1 We raycast the scene in planes that are perpendicular to the X-Z-plane in the world coordinate system. Each plane maps to the screen as a single line.

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

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 are RLE compressed 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 Sect. 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 that share the straight line that passes through the viewpoint and are parallel to the Y-axis (*Down-vector*). Raycasting the RLE data in each concentric plane is performed step-by-step from near to far along the X-Z plane, whereas the rasterization is performed for each step in the vertical direction (parallel to the Y-axis) from top to bottom. More specifically, 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 ray-buffer (a temporary bitmap) for performance reasons. The ray-buffer is then mapped to the screen using the Pixel Shader.

The render loop consists of the following five pipelined 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 (vp) of all concentric planes around the down vector on the CPU. As shown in Fig. 1, the vanishing point is the intersection of the screen plane and down vector. The vanishing point needs to be computed (Sect. 4.1) before the concentric planes (Sect. 4.2).

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 (Sect. 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 view-point'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 (Sect. 4.4.1) at this distance. For each RLE element, we first perform the projection of the coordinates to the ray-buffer (Sect. 4.4.2), then culling is performed (Sect. 4.4.3). Finally, the element is rasterized as a textured line in the ray-buffer (Sect. 4.4.4).

Step 4. Display the ray-buffer on the screen. The GPU Pixel Shader is used to rearrange the rows of the ray-buffer to a radial pattern of straight lines centered at the vanishing point on the screen (Sect. 4.5).

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

3. Pre-Processing

3.1 Organization

The original source data to be visualized can either be volume data or polygon data. In the case of polygonal data, the voxelization is simply performed by rasterizing each triangle as voxels into the 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 RLE. More specifically, each vertical RLE column is compressed separately and referenced by one pointer of a 2D lattice placed in the X-Z plane, where the scale factors of the lattice for the X and Z directions are normally uniform. As shown in Fig. 3, not all the voxels of a solid volumetric object are RLE compressed. To avoid unnecessary memory consumption, only voxel surface data are RLE compressed, whereas occluded voxels in the inner area are removed, i.e., not RLE compressed. Section 3.2 elaborates on the specific data structure of the RLE voxel data.

3.2 Data Structure

The data structure of the voxel data should be able to use 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, and the first (topmost) 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 on the right side in Fig. 4, white voxels indicate skipped voxels and colored voxels indicate drawn voxels. In the leftmost voxel column, the two voxels from the top are skipped (not drawn) and just below there is one (colored) drawn voxel. Therefore, "2" and "1" are stored in the "skipped voxels" field and "drawn voxels" field in the

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.

pointer map element of the RLE structure on the left-hand 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 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' appearances in the RLE column.

For efficient GPU computing, the number of memory accesses has to be minimized. Therefore, we store 64 bit elements in the pointer map because this 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 LOD Computation

As described in Sects. 1.3 and 2, the individual RLE data for each LOD is obtained in advance of the visualization process. We apply the idea of texture mip-maps to the original RLE voxel data and generate RLE mip-volumes. The original RLE voxel data have the highest resolution and are used for the LOD that corresponds to the range closest to the viewpoint. As the distance from the viewpoint grows, RLE voxel data with lower resolutions are used.

More specifically, suppose that lev denotes a LOD, 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 long as that in the level $lev - 1$, where linear down-sampling is applied to the voxel data in

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.

the level $lev - 1$ so that the voxel data in the level lev are obtained.

For example, an original volume of $16 \times 16 \times 16$ has four mip-volumes: $8 \times 8 \times 8$, $4 \times 4 \times 4$, $2 \times 2 \times 2$ and $1 \times 1 \times 1$. As described below, 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 Sect. 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 (Fig. 1). Each plane is projected onto 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.

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

where d denotes the distance between the camera origin (viewpoint) and screen-plane, and α_p 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. However, this case would cause a division by zero. To avoid this, $vp.y$ is set to a large positive constant if $0 \leq \sin(\alpha_p) < \varepsilon$ and to a large negative constant if $0 > \sin(\alpha_p) > -\varepsilon$, where ε defines a positive value near zero. The vanishing point is projected to the screen space by the following equation:

$$vp_{screen} = A_{cam} \cdot \begin{bmatrix} vp.x \\ vp.y \\ vp.z \\ 1 \end{bmatrix} \quad (2)$$

where vp_{screen} represents the projection of vp to the screen space, A_{cam} represents the 3×4 camera matrix. Each plane intersects the screen as one line originated in the vp_{screen} (Fig. 1).

4.2 Concentric Planes

Since each plane is projected onto the screen as one line that originates in the vp_{screen} , we need to focus on achieving the complete coverage of the screen by the lines originated in the 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 the vp_{screen} , and the angle

Fig. 5 Screen segmentation: vp_{screen} represents the vanishing point, Seg.1..4 refers to the four segments and $border_{1..4}$ to the four borders.

between adjacent borders is 90° . We texture each line included in the left and right (with respect to the 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 other end is the vp_{screen} . We can calculate the number of planes (lines) as indicated below:

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

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

4.3 Plane Parameters

As described in Sect. 2, all the calculations described in the rest of Sect. 4 are executed on the GPU in a parallel manner using multiple threads. The number of simultaneously 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 (Sect. 4.3).

4.4 Rasterizing the Ray-Buffer

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

4.4.1 Traversal per Plane

To rasterize the RLE elements to the 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 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 until it reaches the point that corresponds to the predefined maximal distance from the viewpoint. During this traversal, culling, which is explained in Sect. 4.4.3, is performed for the visibility check.

The traversal is not equidistant since it is often performed in volume visualization. As shown in Fig. 6, the 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 [10]. In Fig. 6, we compare the visual results of the exact traversal and 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.

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

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 predefined distance along the line. From P_0 , the point below the viewpoint, to P_1 , which is away from P_0 by pd on the line, the RLE data (voxels) with the highest resolution are used for the rasterization; similarly, from P_1 to P_2 , which is away from P_1 by pd , the second highest resolution is used, and so on.

4.4.2 Projecting RLE Elements to Ray-Buffer

As mentioned earlier, the visible part of each RLE element is rasterized to the ray-buffer as a textured line, where the x , y and z coordinates of the start point ps and end point pe are the 3D world space coordinates of the particular RLE element. We project ps and pe into the screen space using the camera matrix A_{cam} as follows:

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

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, where the origin is placed at the viewpoint, the Z-axis is a straight line from the viewpoint towards the center of the screen, the X-axis is a straight line towards the origin and parallel to the upper and lower screen borders, and the Y-axis is a straight line towards the origin and parallel to the left and right screen borders. To illustrate the different coordinate systems, we have included Fig. 7. The variables ps_{screen} and pe_{screen} are the 2D ray-buffer coordinates of ps and pe . As described in Sect. 4.2, either the horizontal (x) or vertical (y) components of the start and end coordinates are used for rasterizing the 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. 8.

Therefore, either the horizontal (x) or vertical (y) coordinates of the start and end points are used to define the

Fig. 7 Coordinate systems: (a) World space, (b) Camera space, and (c) Screen space.

Fig. 8 Ray mapping: The upper section shows the ray-buffer with the four segments. The lower half demonstrates how the segments are mapped to the screen.

vertical 1D position inside the column of the ray-buffer. In Fig. 8, Segments 1 and 3 use the horizontal (x) coordinate, whereas 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 Sect. 4.4.3), before the textured rasterization is performed (Sect. 4.4.4).

4.4.3 Culling

As described in Sect. 4, culling only needs to be performed to render the visible pixels of the RLE elements and efficiently skip the RLE elements that are entirely occluded. This paper uses three culling methods including novel and known methods that are illustrated in Fig. 9: (a) The hidden line removal algorithm [13], which is also known as the floating horizon algorithm. We implement a modified version of this algorithm, which suits our needs better. (b) Shared memory culling. We use the GPU's shared memory as a binary bitmap to remember which pixels have already been drawn. (c) Per pixel forwarding. This method stores a pointer for each pixel, which points to the next undrawn pixel. This approach was invented by Lacroute [2].

However, using all the algorithms simultaneously is not efficient due to mutual interference. Our tests have shown that it is efficient to use the floating horizon algorithm together with shared memory culling or per pixel forwarding.

a) Floating Horizon / Hidden Line Removal

b) Shared Memory Culling

c) Per Pixel Forwarding

Fig. 9 Culling: (a) Floating Horizon: Store min/max Y values for each column on the screen. (b) Shared Memory Culling: The GPU shared memory serves as a cache for set pixels. (c) Skip Buffer: Each element in the skip buffer stores the number of drawn pixels that can be skipped until a free pixel is found.

Shared memory culling and per pixel forwarding interfere with each other because they are both executed at a per pixel level.

(1) Modified Floating Horizon Culling

We use the well-known floating horizon algorithm [13], Fig. 9 (a), which has already been used in the original voxel forward projection algorithm [7]. The algorithm works as follows:

For each rendered plane, we store two vertical offset values: one start and one end offset along the projected line in the screen, defining the bounds of the renderable 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 us 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 the 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. This modification is as follows: after one RLE element that touches either border is rasterized, we update the offsets and further test the pixels next to the new offsets if they have been drawn already. If they have been drawn already, we further narrow the boundaries. Depending on the scene, this modification can double the algorithm performance.

(2) Shared Memory Culling

The shared memory culling algorithm, Fig. 9 (b), takes ad-

vantage 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 fits entirely into the graphic card's shared memory. Our target hardware, the NVidia GTX series, provides two main types of memory: global memory and shared memory. The difference between them is that memory access to global memory consumes about 300 processor cycles, whereas access to the shared memory only requires one cycle. Therefore, by using a binary visibility map stored in the shared memory, we can quickly apply per pixel culling without accessing the slower main memory. Shared memory culling accelerates the rendering speed by 40% to 140%, depending on the scene.

(3) Per Pixel Forward Culling

Lacroute's culling based on per pixel forwarding [2], Fig. 9 (c), is slightly slower and more complex than shared memory culling. However, 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 ray-buffer we store a relative jump offset. This offset is set to zero at the beginning and is updated once an RLE element is drawn as shown in Fig. 9 (c). Since relative jumps help to skip pixels efficiently, we achieve a speed of approximately 1.08 to 2.0 times, which is significantly faster than the floating horizon algorithm alone, but approximately 20% slower compared with shared memory culling.

4.4.4 Drawing RLE Elements as Textured Lines

Each RLE element is rasterized into one or multiple columns of the ray-buffer as a texture mapped line using the coordinates of p_s and p_e 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 Sect. 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 the correct result.

4.5 Displaying the Ray-Buffer

We can map the texture stored in the 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 (x_s, y_s) on the screen. The mapping is applied in a concentric manner with respect to the vanishing point (vp) as shown in Fig. 8. We

define the formula to compute the source (U, V) texture coordinates inside the ray-buffer by Eq. (5) as follows.

$$\begin{aligned} U_{2,4} &= (x_s - vp \cdot x) \cdot |y_s - vp \cdot y| + s_{2,4} \\ V_{2,4} &= y_s - vp \cdot y \\ U_{1,3} &= (y_s - vp \cdot y) \cdot |x_s - vp \cdot x| + s_{1,3} \\ V_{1,3} &= x_s - vp \cdot x \end{aligned} \quad (5)$$

In the formula, U defines the horizontal coordinate inside the ray-buffer, V the vertical coordinate, x_s the horizontal screen coordinate, y_s the vertical screen coordinate, and s the start offset that is added to 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 (x_s, y_s) range from 0 to 1.

4.6 Quality Aspects

As shown in the flowchart 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 the 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.

4.6.1 Smoothing

We apply smoothing as a post-process in image space by using Pixel Shader. We have developed a special smoothing method that achieves two types of smoothing in one Shader pass: the smoothing of voxel silhouettes and the smoothing of voxels close to the camera themselves. Figure 10 shows an example of the result of this method.

The smoothing contains multiple steps, as illustrated in Fig. 11. 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 5×5 pixels. The scale factor of the box filter is determined by the previously computed minimum depth. For the smoothing, we only include pixels that are within a limited depth range near to the minimum depth. As a result, we can see that both

Fig. 10 Smoothing: The left image shows the basic rendering. To improve the rendering quality, we smooth the silhouette (middle image) and the interior (right).

Fig. 11 Smoothing steps: (a) Target pixel, (b) Find minimum depth (Z), and (c) Box-filter with threshold; scaled according to the minimum depth, (d) Result.

Fig. 12 Anti-aliasing (AA): We investigated three quality levels: No AA (left), 2×1 AA (center), and 2×2 AA (right).

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 to obtain the target resolution. Figure 12 compares three configurations: no AA, 2×1 pixels AA, and 2×2 pixels AA. Obviously, 2×2 pixels AA and 2×1 pixels AA give the best and second best quality.

4.6.3 Screen Space Normals (SSN)

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

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

where x_s and y_s represent the horizontal and vertical coordinates of a pixel in the screen, respectively, $\text{Depth}(\dots)$ represents the depth of the pixel (argument) in the depth buffer, and rnd is the random function to achieve an averaged result for multiple samples, x for computing n is a vector cross-product. We have included Fig. 13 (a) for a better understanding of the variables. Note that Eq. (6) indicates that the sample region needs to be reciprocal in size to the sampled depth value z_s of the pixels (x_s and y_s). In case the

Fig. 13 SSN: (a) depth-buffer sampling, (b) normal vectors and (c) shaded result, enhanced with ambient occlusions.

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. We can see the resulting normal map in Fig. 13 (b). Since computing a random value by GPU is slow, we sample a random value from a texture as an alternative. As Screen Space Ambient Occlusion (SSAO) [11] samples the depth buffer in a similar way to SSN, it is possible to efficiently combine both methods in only one Shader Pass. An example of the result is demonstrated in Fig. 13 (c).

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. 14. Our experimental system consists of a Pentium-D 3.0 GHz Processor with 1 GB of RAM and a GeForce285 GTX (1024 MB) graphics board with 240 stream processors. As shown in Fig. 2, we used NVidia CUDA to compute the raycasting part of the algorithm, whereas texture mapping the ray-buffer and the post-processing were executed in the Pixel Shader. The render resolution for all the tests was set to 1024×768 pixels, whereas the AA setting for improving quality was 2×1, which provided the best tradeoff between quality and performance.

Table 1 shows the result of the 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 bits only for the tree leaves, which sums to about $32 \cdot (1 + \frac{1}{8} + \frac{1}{64} + \dots) = 36.8$ bits for the entire tree. However, in some scenes our algorithm requires more memory than splatting-based algorithms such

Fig. 14 Test scenarios: Handcrafted mansion (upper left), Bonsai forest with 3,000 trees (upper right), Happy Buddha (middle left), and a procedural landscape with about 4,000 visible trees (middle right), the Stanford Dragon and the Stanford Bunny.

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

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

as QSplat, which only uses 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 image quality.

To measure and evaluate the rendering speed, we first investigate the maximum polygon performance of our graphics card. In case of rendering as a quad by two textured triangles, 350 million triangles per second are the limit of our graphics card for rendering triangle strips, whereas splatting reaches 100 million primitives (splats) per second. Table 1 shows 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 is 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 (resolu-

Fig. 15 GPU vs. CPU: We compared our GPU implementation running on an NVidia GTX 285 to the CPU (Intel Q6600 4×3 Ghz).

tion). For testing the performance in case of large outdoor areas, we created scenes containing more than 1,000 instances of the same dataset for the procedural and the bonsai scene. The maximal view distance was set to 40,000 voxels in both cases. The large outdoor scenes of the bonsai and the procedural datasets contributed to Table 1 as well. They were used in all our tests to evaluate performance, the compression ratio, and quality.

To compare the performance of our GPU implementation with the CPU, we also implemented our method for the CPU. 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 4×3 Ghz and 1 GB ram. The GPU outperformed the CPU by a factor of three for light scenes without AA and seven for complex scenes with AA enabled, where the scenes used for testing are shown in Fig. 15.

We further evaluated the render speed in regards to the image quality by measuring the performance for different quality settings. We compared no anti-aliasing, 2×1 anti-aliasing, and 2×2 anti-aliasing (Fig. 12). If the speed for the no anti-aliasing is 100%, 2×1 AA and 2×2 AA achieved 104% and approximately 80%, respectively. The increase in speed for 2×1 AA might be explained as better coalescence for reads from GPU memory. On the GPU, coalescent memory reads are very important. This also means that half the GPU's processing units are idle for the no AA configuration because 2×1 AA requires twice 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 our case, the speed of the memory is the limitation. We have already reduced the memory bandwidth by employing multiple culling algorithms – yet it remains the limiting factor. To further improve on that, additional compression schemes might be helpful.

As shown in Fig. 16, our algorithm can achieve high quality renderings for a scene with many fine structures. To facilitate the comparison, we render the result using 2×2 AA in the left half and no AA in the right half.

Finally, we visualized the Richtmyer-Meshkov dataset, which is one of the largest datasets, with a resolution of 2048×1920×2048. 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 regards

Fig. 16 Quality: To show the ability to render at high quality, we created a complex test scene with many fine details rendered at 512×384 pixels and 2×2 AA as well as no AA for comparison. Note that 2×2 AA successfully removes aliasing artifacts for distant pixels.

Fig. 17 It is possible to render the complete Richtmyer-Meshkov dataset more than 100 times at interactive rates at a 1024×768 screen resolution. For the shading, we used a combination of SSAO and SSN.

to the binary volume data. Since this particular dataset is very large, we do not store color or shading information. For 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 1024×768 is interactive frame rates: 15 fps for rendering a single instance of the dataset (Fig. 15) and 10 fps for rendering the dataset repeatedly as shown in Fig. 17.

6. Conclusions

This paper presents an efficient adaptation of the voxel forward projection algorithm using recent graphics hardware. Several improvements to speed up the algorithm, optimize memory consumption, and improve the rendering quality were proposed. Initially, the surface data are RLE compressed to allow fast decoding on the GPU. According to [12], 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. Furthermore, we were able to visualize large and highly detailed datasets 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. 10 and Fig. 16). In addition, the storage per voxel is significantly less than conventional tree-based raycasting (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 2×1 full screen anti-aliasing at full speed. Using 2×1 AA does not decrease the speed compared with not using AA.

Although the results are 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 support during the initial phase of this research, Mark Duchaineau for providing us the Richtmyer-Meshkov instability dataset 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] 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
- [2] P.G. Lacroute, "Fast volume rendering using a shear-warp factorization of the viewing transformation," SIGGRAPH '94, pp.451–458, 1994.
- [3] A. Knoll, I. Wald, S. Parker, and C. Hansen, "Interactive isosurface ray tracing of large octree volumes," IEEE Symposium on Interactive Ray Tracing, pp.115–124, 2006.
- [4] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering," ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D), pp.15–22, 2009.
- [5] S. Rusinkiewicz and M. Levoy, "Qsplat: A multiresolution point rendering system for large meshes," SIGGRAPH '00, pp.343–352, 2000.
- [6] E. Gobbetti and F. Marton, "Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms," SIGGRAPH '05, pp.878–885, 2005.
- [7] J.R. Wright and J.C.L. Hsieh, "A voxel-based, forward projection algorithm for rendering surface and volumetric data," Visualization '92, pp.340–348, 1992.
- [8] K. Silverman, Voxlap engine, 1999–2003. <http://advsys.net/ken/voxlap.htm>, visited Feb. 2010.
- [9] NVidia Corp, "Compute Unified Device Architecture" (CUDA) <http://developer.nvidia.com/object/cuda.html>

- [10] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing," Eurographics '87, pp.3–10, North-Holland, 1987.
- [11] M. Mittring, "Advanced real-time rendering," in 3D Graphics and Games Course, Chapter 8, pp.113–115, SIGGRAPH, 2007.
- [12] P. Komma, J. Fischer, F. Duffner, and D. Bartz, "Lossless volume data compression schemes," SimVis 2007, pp.169–182, 2007.
- [13] T.J. Wright, "A two-space solution to the hidden line problem for plotting functions of two variables," IEEE Trans. Comput., vol.22, no.1, pp.28–33, 1973.

Sven Forstmann was born in Konstanz, Germany on January 26, 1977. He was awarded an M.S. degree in Computer Science from Karlsruhe University in 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 PhD course at the GITS faculty in 2005. His specializations are in the areas of computer graphics and image processing.

Jun Ohya received B.S., M.S. and PhD 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.