IEEE/EG International Symposium on Volume Graphics (2010)
R. Westermann and G. Kindlmann (Editors)

Multi-layer Volume Ray Casting on GPU

Wei Li[1]

[1]Siemens Corporate Research
Princeton NJ, USA

Abstract

We present multi-layer volume ray casting that integrates various volume rendering enhancements into a unified framework, including empty-space skipping, volume clipping, embedding opaque and semi-transparent objects, volume editing through constructional solid geometry (CSG) operations, etc. The central idea is to consider all these objects as volume-of-interests(VOIs). Each VOI is assigned a priority number to determine how the overlapped regions are handled. All the information of the VOIs are encoded into a ray-layer buffer through depth peeling combined with layer simplification. Each pixel of a ray-layer buffer contains the information of a set of ray segments, including the starting locations and the material IDs. The multi-layer ray caster then renders (or skips) each ray segment with the proper viewing parameters depending on the material IDs. The ray-layer buffer is also used to estimate the gradients of surfaces between the layers to improve shading.

1. Introduction

GPU-based volume rendering has become the most popular volume visualization approach due to the ever-increasing power and programmability of commodity graphics hardware. Especially in recent years, volumetric ray casting on GPU has attracted great attention because of its flexibility to incorporate enhancements to the basic algorithm that either improve rendering speed and image quality or highlight interesting features. In this paper, we present multi-layer volume ray casting that provides a general framework supporting various kinds of such enhancements.

Besides a volume dataset, the multi-layer ray caster also takes a set of VOI models as input. Each VOI defines a sub-region, either connected or disconnected, inside the volume space, and specifies appearance settings for the sub-region. Multiple VOIs can overlap in space. Therefore users may assign different priority numbers to VOIs to resolve any conflict. Please note that the meaning of VOI in this paper is much wider than standard volume-of-interest. We treat various objects as VOI, although they appear very different from standard ones. These objects include clipping geometry, such as cut planes and crop boxes, embedded opaque and semi-transparent objects, editing masks, embedded MPRs (multi-planar reformatting), bounding geometry of non-empty spaces, etc.

To render a volume dataset that is partitioned by multiple overlapped VOIs with different priorities, we first convert all the information of the VOIs into a ray-layer buffer. The ray-layer buffer is generated by depth peeling all VOI boundaries, followed by organizing the peeled fragments into layers. Each pixel in the buffer encodes the information of a set of non-overlapping ray segments. The information includes the starting location and material ID of each ray segment. The multi-layer ray caster then casts rays segment-by-segment according to the buffer. The layer buffer is also used for estimating gradients on layer surfaces to improve shading effects.

In the remainder of the paper, we first list previous publications that directly relate to our work. Then we discuss in more details about volume rendering features that can be rendered as VOIs. Next we present the structure of the ray-layer buffer and its generation, followed by the multi-layer ray casting. Finally we conclude the paper with experimental results and discussions.

2. Related work

As mentioned before, GPU-based volume rendering [HKRs*06] has been widely explored, and there are plenty of publications on this topic. We only discuss literatures that are closely related to our work. The idea of using different parameters and modes across ray segments is adopted from two-level volume rendering [HMiBG01] [HBH03] that fo-

© The Eurographics Association 2010.

DOI: 10.2312/VG/VG10/005-012

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

cuses on segmented volumes with volumetric masks. Hadwiger el al. [HBH03] also presents methods for filtering mask boundaries. In contrast, our approach works most efficiently with geometric masks that are continuous representations of boundaries and do not have to be filtered. Raman et al. [RMC08] replaces some of the 3D volume rendering with 2D image operations to simulate the two-level effects. In their work a layer refers to a Photoshop-like 2D image whereas in our work, the same term is a group of ray segments.

Krüger and Westermann [KW03] render the front and back faces of the bounding box of a volume to generate ray directions and texture coordinates. Scharsach [SHN*06] refines the bounding geometry from the whole dataset to the non-empty regions, hence incorporates empty-space skipping into the ray texture, instead of relying on a separate volume texture [KW03]. This uses the same principle of polygon assisted ray casting [ASK92]. Both these approaches create auxiliary 2D textures that contain the starting and ending locations of rays, which have certain similarity to our ray-layer buffer. However, both these approaches store 3D texture coordinates in the ray texture, whereas our ray-layer buffer only saves the Z value of the starting position of a new layer. Not only does our method require less memory, but it also fulfills the requirement posed by bricking that the sampling locations in different bricks are aligned. Krüger et al. and Scharsach et al. only output the starting and ending texture coordinates for each ray, which essentially is a single layer description in our multi-layer notation. For this reason, the approach of Scharsach el al. can't skip empty spaces between the first and last visible samples. Apparently, the single layer representation can't cope with embedded objects that produce multiple layers. Stegmaier et al. [SSKE05] have reported a single-pass ray caster exploiting dynamic branching and larger number of allowable iterations in the fragment pipeline of the GPU. Except for using a ray-layer buffer, our code for casting a ray segment is similar in structure to their implementation. But beyond that we have an additional outside loop to go through all the non empty ray segments.

There have been a few volume rendering techniques that involve depth peeling. Nagy and Klein [NK03] apply volumetric depth peeling to visualize the first n iso-surfaces. Rezk-Salama and Kolb [RSK06] partition a volume into several view-dependent layers considering the accumulated opacity. Both methods perform peeling on the volume properties, whereas our depth-peeling converts VOI information to a ray-layer buffer. It should be noted that our framework can easily incorporate these kinds of peelings within each ray segment. Brecheisen et al. [BPVtHR08] perform depth peeling on multiple intersecting volumes, and raycast each layer in a separate rendering pass. In contrast, we store the peeling information in a ray-layer buffer, hence only need a single pass for raycasting which is more efficient for GPU hardware. More importantly, the usage of the ray-layer buffer permits the integration of VOI priorities and layer simplification which are difficult for Brecheisen et al.'s method. Lauer and Seiler present multiple-depth-buffer for embedding translucent objects [LS]. The idea of partitioning volume space into layers is similar, but they don't deal with overlapped regions with different priorities.

GPU accelerated CSG modeling, e.g. [LF02] [GKMV03] [HR05], has certain similarities to the generation of the ray-layer buffer, in that they both handle the Boolean operations of multiple geometric meshes. The difference is that CSG literatures focus more on dealing with complicated CSG trees, whereas in our ray-layer buffer generation, the VOIs have different priority numbers that are critical in the interaction between VOIs. At the same time, there can be multiple VOIs with identical priority numbers, while their logical relation is usually much simpler. Therefore our ray-layer buffer algorithm concentrates on how to handle the priority numbers and the material IDs associated with the VOIs. Weiskopf et al. [WEE02] achieve volume clipping through GPU stencil test. We consider clipping as a special case of VOI and integrate it into our multi-layer framework.

Multi-layer volume ray casting supports interactively changing shape and appearance of one or more VOIs, which has certain visual similarity to direct volume editing [BKW08], where they use a volumetric scalar field to hold the output of editing. They also exploit surface-aligned clip geometry to achieve cutaway views. In comparison, our editing masks and VOI masks are preferably geometric meshes. Please also note that our work focuses on rendering, instead of editing, and we have no difficulty in incorporating a volumetric mask in the multi-layer ray casting pipeline.

Kainz et al. [KGB*09] combine multiple volumes and translucent polyhedral objects by ray-casting layers too. Taking advantage of Cuda, they achieve layer sorting by a single-pass polyhedral rendering. However Cuda still lacks certain capabilities, such as texture fetching with trilinear interpolation, that greatly degrades its performance for graphics applications. Moreover, in our experiments, the cost of the multi-pass depth peeling is nearly ignorable comparing with the ray casting. Therefore our strategy is to keep the ray-casting loop as simple as possible and to shift workloads to the layer sorting. Another difference is that our system is oriented towards prioritized VOIs, whereas Kainz et al. are more focused on the blending of different objects.

3. Volume-of-interest

During volume rendering, each voxel is assigned a color depending on its properties, such as the density and the gradient. Then the color values are composited into a frame buffer to form an image. The color assignment typically involves a mapping from voxel properties to an entry in a color lookup table, referred to as a transfer function. In addition, local gradients of voxels plus parameters of lights are used to produce lighting effects that are added to the transfer-function-mapped color.

© The Eurographics Association 2010.

In standard volume rendering, the same computation of color assignment is performed for all voxels uniformly. Many improvements have been proposed. For example, a user may specify a sub-region of the volume space, and voxels inside the sub-region are rendered differently from voxels outside. Such a subregion is referred to as a volume-of-interest (VOI).

More than one VOI can be specified for the same dataset, and multiple VOIs may overlap in space. Each VOI is assigned a priority that is used to determine how the overlapped regions are rendered. A practical policy is that a high-priority VOI overwrites low-priority ones, and we follow this policy in the paper. If an overlapped region is enclosed by two or more VOIs with different viewing parameters but identical priorities, then the color in the region is a blend of the contributions from these VOIs. The bounding box of a volume is actually a VOI with the lowest priority, which we refer to as the default VOI. Any voxel excluded by all other VOIs are rendered according to the default VOI.

Next, we list several VOI variations. Some of them are disguised in forms that are significantly different from the standard VOI definition, and are not considered as VOIs by previous volume rendering approaches.

3.1. Non-empty-space boundaries

Empty space skipping is an important acceleration technique for volume rendering. As suggested by the name, a volume is partitioned into regions. A region containing voxels that are all mapped to fully invisible colors is completely skipped and the image quality is not degraded at all. Non-empty-space boundaries separate non-empty from empty regions. For efficiency, they don't have to be accurate. But all the non-empty voxels should be enclosed. In other words, empty voxels could be included in the non-empty region, but not vice versa. Apparently, non-empty-space boundaries refine the default VOI.

3.2. Cut planes, crop box, and opaque objects

Adding a cut plane is equivalent to defining a VOI as a half space, and voxels inside the VOI are set to be invisible. A crop box is a box that is aligned with the major axes of a volumetric dataset. Intuitively, all voxels outside the box are cropped. According to our VOI definition, the interesting region is outside the box where it should be rendered as empty, whereas the region inside the box is left to other VOIs.

The standard way of embedding opaque objects is that these objects are rendered before volume rendering, so that the depth buffer maintains the depth of the front faces of the opaque objects. In our multi-layer model, a depth buffer containing the depth information of opaque objects is considered as a VOI, because it makes all the volume samples behind its front surface invisible.

3.3. Editing mask and VOI mask

Usually, a mask in volume rendering is given in one of two forms, 1) a polygonal mesh separating interesting and uninteresting regions; 2) a volumetric mask defining the belongingness of each voxel. There are also two main sources for masks. The first one is through imaging operations, such as segmentation, where the output is usually in the form of a volumetric mask. Another source is the editing of a user who draws geometric shapes or tags individual voxels, whose output can be of either form. One editing process in commercial software is called punching, in which a user draws a closed 2D curve that is extruded to form a 3D mesh, and uninteresting portions are "punched away".

For efficiency, the multi-layer ray caster differentiates masks according to the following criteria: whether any voxel selected by the mask is visible. If all the voxels are invisible, we call it an editing mask, otherwise it is referred to as a VOI mask. A voxel selected by a VOI mask means that the voxel is of interest, and not only should it be visible in the rendered image, but it may also be highlighted with a transfer function different from the one of the default VOI. In contrast, a voxel inside an editing mask means it is uninteresting and should be invisible to reveal features behind it.

3.4. Embedded semi-transparent objects

The behavior of a semi-transparent object is similar to a VOI in that it does not change the rendering calculation outside the object. We are most interested in embedding MPR and its variants. Embedding MPRs in volume rendering facilitates an observer to perceive the location and orientation of the MPRs. Please note that all MPRs map to thin surfaces and the volume that they occupy is zero. In other words, the front and the back faces of the VOI defined by a MPR are always at the same depth.

4. Generating ray-layer buffer

Before the actual ray casting, all the VOIs are processed to resolve overlapping and to remove redundant VOI boundaries. The information about the simplified VOIs is written to an intermediate buffer which we refer to as a ray-layer buffer.

4.1. Logical relationship of layer contributors

Without loosing generality, we assume the following order of VOI priorities: cut plane = crop box = background depth buffer > MPR > editing mask > VOI mask > non-empty-space boundaries. We are certainly not interested in invisible spaces, therefore, we consider the complements of cut planes, crop boxes and background depth buffer, referred to as $\overline{cutPlanes}$, $\overline{cropBox}$, and $\overline{depthBuffer}$ respectively. Following is the formula for combining all VOIs:

$$\overline{cutPlanes} \wedge \overline{cropBox} \wedge \overline{depthBuffer} \wedge (MPRs \vee$$

© The Eurographics Association 2010.

$$\overline{editingMasks} \vee voiMasks \vee nonEmptySpaceBoundaries) \quad (1)$$

where $\wedge$ represents *AND*, and $\vee$ denotes *OR*.

Note that each entry in formula 1 may stand for multiple VOIs. For example, *editingMasks* denotes the combination of multiple editing VOIs through user specified Boolean operations. Specifically, during the punching operation described in section 3.3, usually a user enlarges the excluded regions by drawing planar curves at different viewing directions. Therefore, these editing masks are combined through Boolean *OR*.

4.2. Ray-layer buffer

A ray layer-buffer is a screen-sized 2D image, with each pixel corresponding to a ray. Every entry in the buffer contains a depth value specifying where a new ray segment begins, and its material ID. A ray segment ends where a new segment begins. All the samples of a ray segment share the same material ID and adjacent ray segments always have different material IDs. All VOIs considered in our algorithm contribute four types of material IDs:

- mtIdVolume: a ray segment is rendered using the default transfer function and other viewing parameters.
- mtIdEmpty: a ray segment is completely invisible and can be skipped.
- mtIdVOI*i*: there can be more than one VOI masks, each has a unique material ID. *i* is the mask index.
- mtIdMPR: representing all types of MPRs.

Each VOI is rendered with an ID. Some use the corresponding material ID, while others are associated with intermediate mesh IDs that eventually are converted to material IDs. There are two such mesh IDs: meshIdClipping and meshIdMprClipping, both involve clipping. Following is a mapping of VOIs to material IDs and mesh IDs:

- Bounding box of non-empty spaces: mtIdVolume
- Cut planes, crop box, background depth: meshIdClipping
- Editing mask: mtIdEmpty
- Voi mask*i*: mtIdVOI*i*
- MPR: mtIdMPR
- MPR with clipping front enabled: meshIdMprClipping

The fragments resulted from rendering the VOIs are depth-peeled into layers and then merged according to their priorities. Figure 10 shows an examplery ray-layer buffer of a single ray. Figure 10(a) shows the VOI regions that intersect this ray, including two parts from the bounding boxes of non-empty spaces, one cut plane, one editing mask, one VOI mask and one MPR. The heights of the VOI boxes in 10(a) represent their priorities. Figure 10 (b) shows ray layers after resolving all overlapping using the priorities of the VOIs. The information of (b) is encoded in the ray-layer buffer displayed in (c), in which the name of each box is the material ID omitting the mtID prefix. The arrows pointing from (c) to (b) illustrate the starting depths of the ray segments.

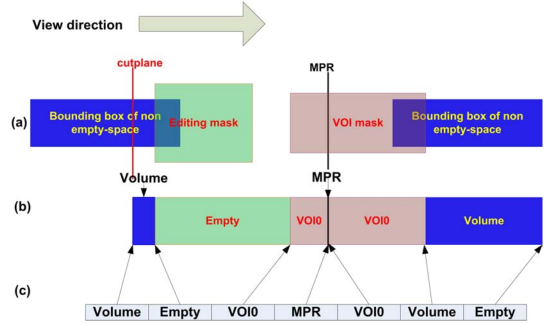


Figure 1: *Ray-layer buffer of a single ray. (a) Contributing VOIs. The heights of VOIs represent their priorities. (b) Merged VOIs with overlapping resolved. (c) Ray-layer buffer where the name in the boxes are material IDs and the arrows to (b) illustrate the starting depths of the ray segments.*

Because our renderer supports bricking, the starting location of each ray segment does not need to be very accurate, as it will be rounded to the nearest aligned sampling location anyway. Taking advantage of this property, we encode the depth and material ID of each ray segment into a single floating point number: $rayInfo = depth + beo \times mtId$, where *beo* is the boundary encoding offset, and equals to the maximal depth value. If the accuracy of the depth becomes a concern, they can certainly be stored in a separate buffer at the cost of increased memory.

For GPU implementation, the layer information is packed into the color channels of multiple RGBA textures. In most of the cases, the maximal number of layers *maxLayers*, and the total number of color channels *numColorChannels* conform to this equation: $maxLayers = numColorChannels - 1$. A special case is when there is an MPR that does not intersect any other VOIs. Then the corresponding MPR layer is represented by a single entry in the ray-layer buffer. In the extreme case, there could be *numColorChannels* MPR layers in a scene.

4.3. Layer generation

A ray-layer buffer is generated by depth-peeling [Eve01] VOI definitions. The peeled fragments are then grouped into ray layers. In GPU-based CSG, a stencil buffer is employed to count the difference between the number of front-facing fragments and that of back facing fragments, in order to determine whether a fragment is enclosed in a primitive. Unfortunately, it is difficult to adopt this approach, because each of our VOI is associated with a priority and an ID, that are critical in layer placement. In addition we need to apply optimizations to reduce the complexity of layers. Therefore, we perform similar stencil tests inside a GPU program.

Figure 2 is a block diagram of layer generation from peeled fragments. We maintain an *insideFlag* array, that

© The Eurographics Association 2010.

counts the front and back-facing fragments of each material ID, even if a fragment is discarded and does not contribute to the layer buffer. By scanning the *insideFlags*, the material ID of the VOI that encloses the input fragment and has the highest priority is found. Denote it as *currentId*. If *currentId* is different from the material ID of the previous layer, the current fragment is written to the buffer with *currentId*. Note that a back-facing fragment indicates the exit of the contributing VOI, therefore *currentId* will be different from the ID of the VOI. For example, in Figure 10(c), the start of the right most volume layer is contributed by a back-facing fragment of the VOI mask, but the material ID is *Volume*. Clipping mesh either clears all previous layers if it is front facing, or marks the layer generation as finished for this ray if back-facing. MPR meshes are simply written out without further process. If the MPR is currently inside an non-empty layer, an entry with the same depth as the MPR but using the previous material ID is appended so that the layer being intersected continues. In Figure 10(c), the MPR and the second VOI0 represent such a case.

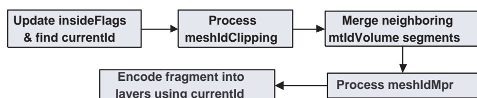


Figure 2: *From peeled fragments to layers.*

One optimization in layer generation is dedicated to the bounding boxes of non-empty spaces. As we know, the union of the bounding boxes is a hull of all non-empty voxels, and it may enclose empty voxels depending on the trade-off between the simplicity and the accuracy of the hull. The hull could be further simplified view-dependently during layer generation without affecting image quality. Therefore, if the empty segment between two adjacent mtIdVolume layers is shorter than a given threshold, they are merged into one.

4.4. Fragments with identical depth values

If two fragments falling to the same pixel have identical depths, then standard peeling algorithm only outputs one of them. This produces incorrect *insideFlags* for the pixel. Please note that unless two polygons are co-planar, the problem only happens in rare cases when the common points of the two surfaces project exactly to the center of certain pixels. It is reasonable to require that the boundary of each VOI does not self-intersect. But it is still possible that a back facing surface touches a front facing surface of the same VOI, which is quite common for the corners and edges of non-empty-space bounding boxes. Different VOIs could also produce fragments with same depth and same facing. We need a comparison that considers depth, facing and meshId which is unfortunately unavailable in standard depth test. One solution for current GPU is that all these factors are encoded to form a new number in a similar way to the encoding of ray-layer buffer entries. The encoded number is set as the modified depth in the peeling fragment program. Some hardware may not support depth test anymore if the depth is modified in a fragment program. In that case the value could be written to one of the color channels and use the MIN or MAX operation of the frame buffer.

5. Multi-layer volume ray caster

Our system supports bricking that partitions a volume dataset into multiple small sub-volumes, each called a brick. The ray-layer buffer is created for the whole volume and shared by all bricks. We feel this way is much easier to detect any error or bug in the ray buffer, and our experiments show that it is more efficient than using a different ray-layer buffer for each brick.

5.1. Ray casting using ray-layer buffer

Once the ray-layer buffer is generated, the multi-layer ray caster takes the buffer as input and casts rays segment-by-segment. It first fetches and decodes the information of the current layer, that includes the start (*rayStart*) and end (*rayEnd*) locations of the ray-segment plus the material ID. If it is a valid segment, it calls raycast1Layer to render the segment. raycast1Layer returns a Boolean flag. If true, the ray caster continues to the next layer. A false notifies the main loop that the ray casting is terminated, and no need to process the following layers.

The flowcart of raycast1Layer is shown in Figure 3. The function first checks if *rayStart* is behind the far end of the current brick to determine whether the current and all the following ray segments are outside the brick. If so, raycast1Layer simply returns false indicating the termination of the ray for the current brick. If renderMpr flag is set, an MPR layer is rendered before ray casting in the intersection of the range $[rayStart, rayEnd]$ with the current brick. Then the accumulated opacity is inspected to dertmine possible termination. Whether *rayEnd* is behind the far end of the brick is an indicator of whether all the following layers are outside the current brick. Depending on this indicator, raycast1Layer returns false for termination or true to continue to the next ray segment.

A material ID uniquely determines which parameters are used to render a ray segment. The most common parameter that changes with material ID is the transfer function. To support multiple transfer functions in a single rendering pass, we stack these transfer functions into a single 2D texture, and add an offset equals to $materialId \times D$ to the texture coordinates, where D is the size of a single transfer function texture divided by the size of the stacked texture, both along the stacked direction. The right portion of Figure 12 shows a texture stacking two transfer functions, and the

© The Eurographics Association 2010.

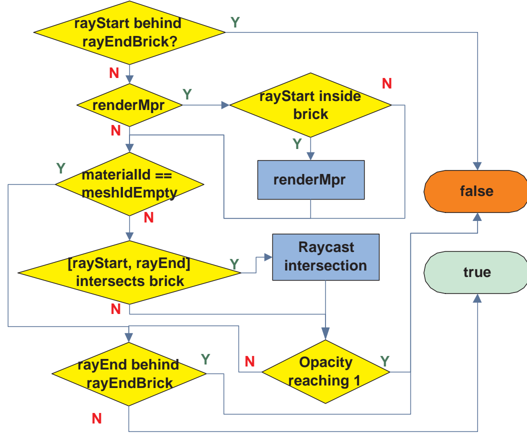
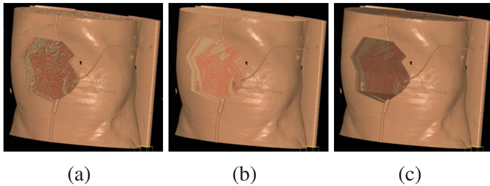


Figure 3: *Ray-cast of a single segment.*

red arrows indicate the transfer function usage of the VOIs. To be more efficient, MPRs and editing masks are handled specially. For a MPR layer, a color is directly assigned in the program instead of going through a texture lookup. The front faces of editing masks are converted to $mtIdEmpty$ during ray-layer buffer generation, and all ray segments labeled with $mtIdEmpty$ are skipped.

5.2. Gradient estimation with ray-layer buffer

The surfaces of a mask could cut through volume regions having low gradient magnitudes. In other words, the quality of gradients in these areas are poor. If shading is performed there, images will be quite noisy, as shown in Figure 11(a). One option is to disable shading in regions with low gradient magnitude, as shown in 11(b). Images then become less noisy, but the mask surfaces look very flat. Figure 11(c) is an image shaded with gradients estimated from the ray-layer buffer. Obviously, the shading provides additional hints on the shape of the mask.



(a) (b) (c)

Figure 4: *Shading using the ray-layer buffer; (a)shading using gradients of the volume; (b)Shading disabled in low gradient magnitude regions; (c)shading with gradients estimated from the ray-layer buffer.*

.

The ray-layer buffer contains sufficient information for computing the gradients of layer surfaces. The main issue is that the ray segments are encoded individually for each ray. For example, a surface falls to layer i of ray $r1$. But the same surface could be stored as layer j for ray $r2$. Where $r1$ and $r2$ are direct neighbors, and $i \neq j$. Therefore, a natural way is to search the neighboring ray segments. The one that has the same material ID and the minimal difference in depth is most likely contributed by the same surface. In practice, we adopt an even simpler approach. It is based on the assumption that the depth values produced by the same surface have strong coherency. An entry in the ray-layer buffer has at least one direct neighbor that is produced by the same surface and is encoded into the same layer. Figure 5 is the flowchart of the simplified gradient estimation. The blue square indicates the entry in the ray-layer buffer corresponding to the layer being ray casted. The four yellow squares represent the entries of the same layer in the four neighboring pixels of the ray-layer buffer. Note that we only take the difference with smaller absolute value to avoid computing gradients using depth values from different layer surfaces. Also note, the computation is performed on entries encoded with material ID, so that differences between entries with different material IDs are huge. $pixelSpacing$ is the distance between two neighboring pixels in world space. Here we assume the image is isotropic in X and Y directions. The surface gradient is then blended with gradient computed from the volume: $gradient = blend * volumeGradient + (1 - blend) * surfaceGradient$, where $blend = min(1, gradientMagnitude/threshold)$.

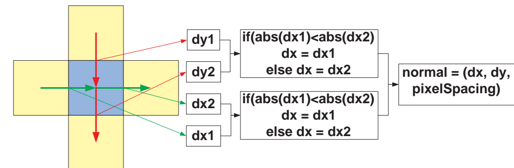


Figure 5: *Gradient estimation from the ray-layer buffer. The blue square indicates the entry in the ray-layer buffer corresponding to the layer being ray casted. The four yellow squares represent the entries of the same layer in the four neighboring pixels of the ray-layer buffer.*

6. Experimental Results

We have implemented the multi-layer ray casting using OpenGL 2.1 and GPU programs written in GLSL. All the images are rendered with a Nvidia Quadro FX 5600. Figure 12 is a screen shot of our testing system, that is configured with three MPR views and one volume rendering (VR) view. The bottom MPR is overlayed with a mask curve editor. Masks are created by extruding from the 2D planar curves. Two mask curves are applied. In the VR image, we can see that the VOI mask is the union the two tube-shaped regions parallel to the directions of head-to-feet and

© The Eurographics Association 2010.

anterior-to-posterior respectively. The transfer function texture stacked with two TFs are shown in the rightmost column with red arrows indicating the regions that are color-mapped by the TFs. Note that the multi-layer ray caster is not limited to extruded meshes at all. It supports arbitrary closed polygon mesh as long as there is no self-intersection. We restricted our test cases simply because such a 2D curve editor is easy to implement and use.

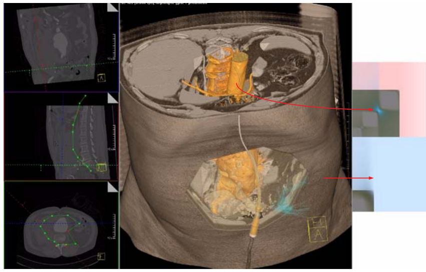
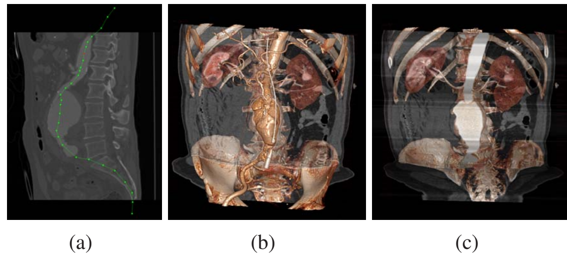


Figure 6: *Screen shot of the testing system for multi-layer ray casting.*

Figure 13 demonstrates the capability of the multi-layer ray casting in handling embedded curve MPR. Figure 13(a) is the 2D curve that is extruded into a 3D surface defining the sampling locations of the curve MPR image. The curved surface is expected to better follow features inside a dataset. 13(b) displays the curve MPR embedded into the volume rendering. 13(c) represents the same scene settings but with all volume samples in front of the surface removed. Note that Figure 13(c) also enables shading on the MPR surface while 13(b) does not.



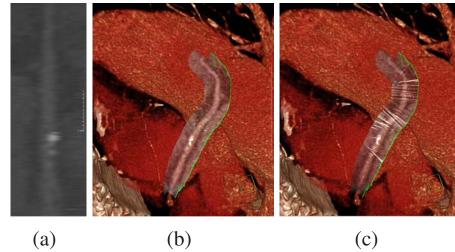
(a) (b) (c)

Figure 7: *Curve MPR. (a) 2D planar curve. (b) Curve MPR embedded into volume rendering. (c Embedded curve MPR with clipping front enabled and shading on the curve surface.*

.

Figure 14 displays the rendering of a ribbon MPR. As suggested by its name, a ribbon MPR is just like doing a MPR along a ribbon that bends and twists in 3D space. Figure 14(a) is the ribbon MPR image, and Figures 14(b) and (c) show the ribbon MPR embedded into volume rendering without and with shading on the ribbon surface respectively. The embedded ribbon MPR clearly conveys the 3D shape and orientation that is lacked in the 2D ribbon MPR image. Note that the shading of the ribbon surface highlights the self-folded areas of the ribbon.

Figure 9 presents the relative performance of the multi-layer volume ray caster to a reference single-layer renderer. The reference renderer is used in a commercial medical imaging software and has been extensiv ely optimized by



(a) (b) (c)

Figure 8: *Ribbon MPR. (a) Normal 2D ribbon MPR. (b) Ribbon MPR embedded in VR. (c) Embedded ribbon MPR with shading.*

.

professionals. The absolute FPS depends on various factors, such as the viewport size, dataset, transfer function, etc. The single-layer renderer uses a ray texture that specifies the start and end position of each ray for empty space skipping. When no VOI is enabled, the multi-layer renderer is slightly slower. When one or more editing masks that remove interior are enabled, the relative speed of the multi-layer renderer vs. the single-layer version becomes faster, and it increases with the number of editing masks. This is because the size of empty spaces increases for the multi-layer ray caster, while the single layer renderer ignores all the masks. With three editing masks, the multi-layer ray caster is 18% faster than the single-layer renderer. The overhead of the ray-layer buffer generation is approximately linear to the number of VOIs. If excluding this overhead, which is 14% for three VOIs, the multi-layer ray caster would be 32% faster because of the skipping of more empty spaces. We also did a test by setting the transfer functions of the masks to be exactly the same as the default VOI, so that the multi-layer ray caster renders identical images as the single-layer ray caster, no matter how many VOIs are enabled. The results are shown in Figure 9 as the serie labeled "‘Relative FPS using VOI masks". Obviously, this setting should not be applied in any practical usage. But the data does show the overhead inside the mulit-layer ray casting GPU program where each ray segment is rendered as multiple smaller ray segments in this scenario, although every segment uses identical parameters. With three VOIs, the overhead in GPU program is about

© The Eurographics Association 2010.

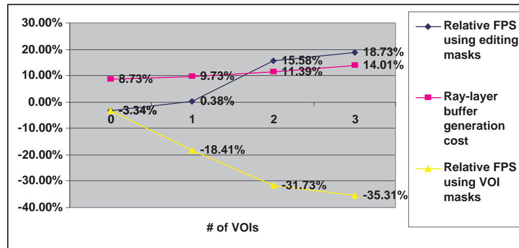


Figure 9: *Relative performance of the multi-layer volume ray casting vs. single-layer ray caster and cost of ray-layer buffer generation*

21%, which equals to the overall slowdown (35%) minus the ray-layer buffer generation cost (14%).

7. Conclusion and discussion

In this paper, we have presented multi-layer volume ray casting. It provides a unified framework for handling various enhancement to standard volume ray casting. In our current implementation, the peeling still requires multiple rendering passes and exhibits unignorable overhead. There have been a few modification proposals for GPU, e.g. [BCL*07], and utilizations of GPU extensions [MB07], so that each peeling generates multiples sorted fragments. The generation of the ray-layer buffer certainly can take advantage of these enhancements if they become available. If in the future, the stencil test becomes more programmable and flexible, there is an option for us to move some functions in layer generation from the fragment pipeline to the stencil operations. Even a simple extension that makes the stencil buffer readable to the fragment program will be very helpful. Fortunately it becomes available in OpenGL 3.0 [SA08]. With this new feature, we can use the stencil tests to count front and back faces of VOIs, instead of tracking these information inside a fragment program, and it avoids the identical depth problem completely. Depending on the contents of the stencil buffer, the encoding fragment program determines the material ID of each layer based on their priorities, and outputs as appropriate to the ray-layer buffer. With the advancement of future GPU, the overhead of multi-layer ray casting will have much less impact, and its advantage will be more obvious.

References

[ASK92] Avila R. S., Sobierajski L. M., Kaufman A. E.: Towards a comprehensive volume visualization system. In *Visualization* (1992), pp. 13–20. 2

[BCL*07] Bavoil L., Callahan S., Lefohn A., Comba J., Silva C.: Multi-fragment effects on the GPU using the k-buffer. In *Interactive 3D Graphics and Games* (2007). 8

[BKW08] Bürger K., Krüger J., Westermann R.: Direct volume editing. *IEEE TVCG 14*, 6 (2008), 1388–1395. 2

[BPVtHR08] Brecheisen R., Platel B., Vilanova A., ter Haar Romeny B.: Flexible GPU-based multi-volume raycasting. *Vision, Modelling and Visualization* (2008), 1–6. 2

[Eve01] Everitt C.: Interactive order-independent transparency. *Nvidia white paper* (2001). 4

[GKMV03] Guha S., Krishnan S., Munagala K., Venkatasubramanian S.: Application of the two-sided depth test to CSG rendering. In *Interactive 3D graphics* (2003), ACM, pp. 177–180. 2

[HBH03] Hadwiger M., Berger C., Hauser H.: High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. *Visualization* (2003), 301–308. 1, 2

[HKRs*06] Hadwiger M., Kniss J. M., Rezk-salama C., Weiskopf D., Engel K.: *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. 1

[HMiBG01] Hauser H., Mroz L., italo Bischi G., Grer M. E.: Two-level volume rendering. *IEEE TVCG 7* (2001), 242–252. 1

[HR05] Hable J., Rossignac J.: Blister: GPU-based rendering of boolean combinations of free-form triangulated shapes. *ACM Trans. Graph 24* (2005), 1024–1031. 2

[KGB*09] Kainz B., Grabner M., Bornik A., Hauswiesner S., Muehl J., Schmalstieg D.: Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore gpus. *Siggraph Asia* (2009). 2

[KW03] Krüger J., Westermann R.: Acceleration Techniques for GPU-based Volume Rendering. In *Visualization* (2003). 2

[LF02] Liao D., Fang S.: Fast volumetric CSG modeling using standard graphics system. In *ACM symposium on Solid modeling and applications* (2002), pp. 204–211. 2

[LS] Lauer H., Seiler L.: Method and apparatus for volume rendering with multiple depth buffers. *US Pathent 6310620*. 2

[MB07] Myers K., Bavoli B. L.: Stencil routed k-buffer. *Nvidia white paper* (2007). 8

[NK03] Nagy Z., Klein R.: Depth-peeling for texture-based volume rendering. In *Pacific Graphics* (2003), p. 429. 2

[RMC08] Raman S., Mishchenko S., Crawfis R.: Layers for effective volume rendering. In *Volume Graphics* (2008). 2

[RSK06] Rezk-Salama C., Kolb A.: Opacity Peeling for Direct Volume Rendering. *Computer Graphics Forum 25*, 3 (2006), 597–606. 2

[SA08] Segal M., Akeley K.: The OpenGL graphics system: A specification (version 3.0), 2008. 8

[SHN*06] Scharsach H., Hadwiger M., Neubauer A., Wolfsberger S., Bühler K.: Perspective isosurface and direct volume rendering for virtual endoscopy applications. In *EUROVIS* (2006), pp. 315–322. 2

[SSKE05] Stegmaier S., Strengert M., Klein T., Ertl T.: A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Volume Graphics* (2005), pp. 187–195. 2

[WEE02] Weiskopf D., Engel K., Ertl T.: Volume clipping via per-fragment operations in texture-based volume visualization. In *Visualization* (2002), pp. 93–100. 2

© The Eurographics Association 2010.