An Interactive Hybrid Renderer using Multi-Layer Depth Maps

Abstract

Current interactive hybrid (combined volume and surface) rendering techniques are only able to embed opaque polygonal objects and therefore cannot provide accurate depth perception of semi-transparent surfaces. This paper proposes a volume rendering approach based on a Multi-Layer Depth Map (MDM) that enables multiple translucent polygons to be embedded. This novel approach generates multiple depth map layers by ray tracing the polygons and sorting distances to intersections. It then blends colors and opacities of polygons and volume segments together in depth order by referencing the depth layers. Our approach is highly scalable allowing thousands of translucent polygons to be rendered in a volume at arbitrary positions in real time. To further improve depth perception, it also supports embedding smoothly shaded and texture-mapped polygons. As an example, the paper provides details for embedding view-dependent text labels in a volume. We present an optimized OpenCL™ implementation of the renderer capable of rendering interactively a volume of 256x256x178-voxels on commodity hardware with 8K polygons embedded, and on a netbook with 1K polygons. We also present screenshots of renderings created by our renderer and by an existing rendering technology to showcase the increased depth perception of the embedded polygons.


1. Introduction

Many visualization problems require both volumetric and polygonal data to be presented in the same scene. For instance, in the petroleum industry, reservoir visualization involves representing wells-bores, pipelines, boundaries and other iconography as embedded polygons in a volume showing geologic information [LLRN09]. In medical imagery, virtual endoscopy requires polygons for displaying endoscopic devices, boundaries and iconography embedded in a volume representing soft tissue, bone tissues and body cavities [SHN’06]. To better assist the decision making process, these volume renderings need to be interactive and scalable, and must encode depth information correctly so that polygons are embedded in the volume. However, existing volume rendering techniques fail short to provide these characteristics. This paper addresses these shortcomings with algorithms that efficiently render scenes with volume and embedded translucent polygon surfaces that run on commodity hardware.

Rendering techniques dealing with mixtures of volumetric and polygonal data can be classified into two groups: approaches using Single-Layer Depth Maps (SDM) and approaches using Multi-Layer Depth Maps (MDM). SDM approaches [SHN’06, Vor, LLRN09, NMHW02, Har02] use a single depth value per pixel for drawing polygons thereby limiting the depth perception of embedded polygons as only those nearest to the camera. In contrast, MDM approaches consider multiple depth values per pixel thereby enabling polygon transparencies to be encoded; this in turn allows to represent all embedded polygons in depth (not only those closest to the camera) and therefore provides correct depth perception. MDM approaches are generally used for surface rendering and none of the existing MDM implementations support all the volume visualization requirements mentioned above.

This paper proposes an MDM approach to render volumes and translucent polygons concurrently at interactive frame-rates. Specifically, the rendering algorithm operates by (i) simultaneously ray tracing through the volume for each pixel while tracking polygon intersections, which are stored in a stack of depth layers and (ii) compositing colors and opacities of volume and polygon samples by referring to the
depth layers to produce the output pixel value. In addition we present an extension of the renderer that enables embedding view-dependent and interactive text labels as textured polygons in the volume; this special case demonstrates how easily our renderer can be extended to support other forms of polygonal data.

We mitigate the added complexity and memory footprint of using multiple depth layers by implementing the renderer on GPU. In particular we propose algorithms to (i) parallelize ray tracing and (ii) accelerate ray-polygon intersection calculations by only considering polygons that are inside the region defined by the pixel in which the ray originates.

To summarize, the major contributions of this paper are:

- Introduce a MDM based volume renderer capable of rendering multiple translucent, arbitrarily positioned polygons and supporting independent shading of polygons and volume.
- Propose an efficient GPU implementation in OpenCL™ of the ray tracer.
- Define an extensible framework that enables embedding other data types in a volume.

The rest of the paper is organized in the following manner: Section 2 gives a brief summary of related work in rendering the mixture of volumetric and polygonal data. Section 3 provides an overview of the rendering algorithm. Section 4 describes the implementation details. Section 5 introduces the algorithm to embed text labels inside a volume as a specific example of embedding textured polygons. Section 6 presents performance analysis and experimental results. Finally, we conclude the paper and describe the future work.

2. Related Work

Most modern volume rendering solutions capable of embedding polygons rely on SDM approaches. They are limited to rendering opaque polygons due to the fact that they use a single depth value per pixel. [LLRN09] proposes a renderer used in the petroleum industry that can efficiently render slices of a reservoir volume by using texture mapping. Polygons are then rendered directly on top of the volume as overlays. [SHN°06] presents an application of virtual endoscopy of an optimized ray-casting pipeline implemented on GPU using two successive loops: a first-hit ray casting [NMHW02] followed by a direct volume rendering ray casting. [Vor] proposes a renderer implemented on GPU using a shading language to render the volume and a single polygonal object (limited to sphere or cylinder) sharing the same shading properties. Color and depth buffers for volume and geometry are stored separately; then colors for both data are composited in depth order by referencing their corresponding depth buffers. [Har02] presents a hardware accelerated implementation using impostors splitting for realistic real-time rendering of volumetric clouds and opaque polygonal objects for flight simulators and games.

Contrary to SDM, MDM approaches have the potential to embed translucent polygons. Marc Levoy [Lev90] pioneered MDM approaches in volume rendering by extending a ray tracer to handle both polygons and volumes. Rays are simultaneously cast through a set of polygons and a volume data array. Samples of each are drawn at equally spaced intervals along the rays, and the resulting colors and opacities are composited together in depth-sorted order. Levoy’s renderer is capable of displaying 3 semi-transparent texture-mapped polygons. In this renderer, all polygons must lie strictly within the boundary of the volume and the CPU based implementation makes it unsuitable for interactive applications on large data sets.

[HKL°97] proposes an MDM implementation able to render volume and translucent object surfaces by sorting polygons and volume sampling slices in depth and subsequently by compositing them from back to front. Although the renderer is accelerated with 3-D texture mapping, it cannot achieve reasonable performance if the embedded objects consist of more than 1K triangles.

Traditional GPU-accelerated MDM approaches found in the literature are geared at polygon rendering. They differ on the algorithm used to perform ray-polygon intersections which in turn are used to generate the depth map layers. To list a few: [CHCH06] introduces a method for fast intersection of triangular meshes by using a threaded bounding volume hierarchy; [YLM06] performs efficient ray-triangle intersection by using a level of detail (LOD) based scheme which integrates simplified LODs of the model into a k-d tree; [PN08] performs fast ray tracing by allowing a group of rays to start traversing the tree data structure from a node deep inside the tree structure; [WBS07] ray traces deformable objects using a bounding volume hierarchy (BVH); [PN08] proposes an algorithm using a three dimensional data structure for ray casting by using a fast multi-split algorithm; [TS05] provides a detailed comparison for three hierarchical scene partitioning structures used to accelerate ray tracing on GPU: uniform grid, kd-trees and bounding volume hierarchies.

3. Rendering Algorithm Overview

As illustrated in Figure 1, the proposed MDM rendering algorithm operates in two steps:

1. Ray-polygons Depth Map calculations: we compute for all rays corresponding to the pixels of the output image, arrays of distances to intersecting polygons, as depth map layers. We further accelerate the ray-polygon intersection tests by only considering polygons that are in a region compatible with the pixel from which the ray originates.

2. Compositing: we ray trace the volume for each pixel of the output image while compositing colors of the volume samples with colors of the intersected polygons as referenced from the corresponding depth arrays.
The creation of the depth arrays is the most time critical part of the rendering pipeline. For interactive applications, where the view is constantly manipulated by the user, the depth arrays need to be refreshed in real time; we therefore rely on the GPU to implement depth array calculations. Compositing is particularly suited to be parallelized as pixels are independent from each other, thus it is also implemented on GPU. In order to support a wide range of graphics hardware our algorithms are implemented using the Open Computing Language (OpenCL™) [Ope10] that defines a cross-platform standard for parallel programming.

4. Renderer Implementation

We start this section by presenting a brief introduction to OpenCL™ that we use extensively to accelerate our rendering algorithms. We then proceed with a detailed description of the outlined rendering steps: ray-polygon depth map calculations and compositing.

4.1. OpenCL™

OpenCL™ is an open industry standard for programming a heterogeneous collection of CPUs and GPUs [Ope10]. From an architecture perspective, OpenCL™ defines devices composed of one or more compute units (CU). Each CU is composed of one or more processing elements (PE). Devices are connected to a single host. An OpenCL™ application running on the host submits commands to be executed on the PEs of each device. A function declared in a program and executed on an OpenCL™ device is called a kernel while a kernel instance is called a work-item. Work-items are organized into work-groups. Work-items in a given work-group execute concurrently on the PEs of a CU.

Different types of memory are defined based on their access model. Private memory can only be accessed by the work-item residing on a given PE. Local memory can only be accessed by work-items of a work-group residing on a given CU. Global memory can be accessed by all work-items on the device. Constant memory is defined by the part of global memory that stays constant during execution. Compared to local memory, global memory provides a much larger storage capacity at the cost of an increased access time. Memory optimizations are most critical for improving performance. One should use as much fast-access memory (i.e. local memory) and as little slow-access memory (i.e. global memory) as possible [Ope10]. The OpenCL™ architecture and memory model is shown in Figure 2.

Figure 1: Overview of the hybrid renderer.

4.2. Ray-polygons Depth Map Calculations

This step takes polygon geometries to compute the distances to all intersecting polygons for each ray. The resulting stack of distances can be thought of as separate depth map layers. A naïve approach for creating these depth layers is to ray trace through all polygons to compute intersections and storing the distances to the intersected polygons. Because of the cost of computing intersections, such an approach would not scale well with the number of polygons and would therefore be impractical for interactive applications.

There are several partitioning schemes and associated data structures that can be used to reduce the number of ray-polygon intersection tests: bounding volume hierarchies [WBS07, CHCH06], BSP trees [IWP08], k-d trees [RSH05, YLM06], uniform grids [PN08], adaptive grids [KS97], etc. We opt for uniform grids because they have constant access time and are particularly well suited for parallelization [PBMH05]. The motivation for partitioning in image space comes from the fact that ray-polygon intersection tests need only to be carried out for the subset of polygons whose projection onto the image is close to the pixel where the ray originates. We propose two scalable optimization schemes that partition the output image area into uniform grids, and then sort the projected polygons into each cell of the grid. Namely:

- **Polygon Driven Approach (PDA)** where one work-item is assigned for one polygon. This approach requires using the global memory for assembling the depth-arrays from each work-item, therefore it is suitable for a large number of polygons.

- **Cell Driven Approach (CDA)** where one work-group is assigned for one cell of the partitioned output image. It can be implemented on local or global memory. For
a small number of polygons (<10K), the greatest per-
formance is achieved on local memory implementation,
which we provide details in the following section.

In practice, the choice of using which approach depends
on the number of polygons and the hardware capabili-
ities. Section 6 provides detailed comparison of the two ap-
proaches.

Structurally, PDA and CDA proceed according to the
same steps. Unique indices are used to track polygons
through these steps:

1. **Polygon Partitioning**: polygons are sorted into image
cells depending on their projected coordinates, resulting
into arrays of polygon indices.
2. **Optimization of Arrays of Polygon Indices**: arrays of
polygon indices are re-organized for efficient access.
3. **Depth Arrays Generation**: ray-polygon intersections
are computed using arrays of polygon indices and dis-
tances are assembled into depth arrays in order.

The purpose of the first two steps is to create an organized
list of polygon indices that is then used to speed-up the depth
array generation. The depth array generation step is common
to both the PDA and the CDA and is realized by ray tracing
from each pixel of the output image and computing intersec-
tions using the algorithm described in [Gla90]. Specifically,
this step uses the polygon index arrays to find possible inter-
sections and stores the distance from the camera to intersect-
ing polygons into the depth array. Depths values are stored
with the same order as the polygon index array; we then re-
order the depth arrays in increasing order using a Bubble
Sort [KS97]. Note that in our implementation the element
of depth arrays records both the depth value and the corre-
sponding polygon index; these indices are used to refer back
to the polygon color when compositing. We detail in the fol-
lowing sections the polygon partitioning and polygon index
arrays optimization steps for the PDA and the CDA.

### 4.2.1. Polygon Driven Approach

This approach is illustrated in Figure 3.

**Polygon Partitioning** In this approach each polygon is
associated with a work-item. First, the kernel creates a local
memory array mapping to all image cells and initializes it
with unset values (typically -1 if polygon indices start at 0).
Then, it loads the polygon’s geometry from global memory
to private memory and projects the vertices in the image via
the model-view matrix. Finally, the kernel updates the index
array with the polygon index if the polygon projects in the
cell. This process is illustrated in Figure 3 (b) where the col-
ors in the output image denote the image cells.

**Optimization of Arrays of Polygon Indices** Arrays of
polygon indices of the work-items are arranged into a com-
 pact, access optimized structure. This process is illustrated
in Figure 3 (c) and (d): local memory arrays are stacked to-
gether by a copy operation into global memory while the
count of polygons per cell row is computed and stored in the
first array element; all the indices are then compacted at the
start of each row.

**Figure 3**: Polygon Driven Approach to ray-polygons depth
map calculations. (a) The output image is segmented into
cells shown as colored areas. Polygons are labeled with
unique indices 1 to 4. (b) Index arrays in local memory of
each work-item resulted by the polygon partitioning step. (c)
Stacked index arrays in global memory with polygon count
per cell. (d) Compacted index arrays in global memory.

### 4.2.2. Cell Driven Approach

This approach is illustrated in Figure 4.

**Polygon Partitioning** In this approach each image cell is
associated with a work-group. Each work-group iterates over
cell polygons and records the index of the polygon whose pro-
jection falls into its cell. These work-groups can contain one
or more work-items. In our implementation we use as many
work-items as allowed by the size of the local memory; if
there is no local memory limitation then the optimal config-
uration (i.e. for which all the polygons are processed in par-
allel) is to have one work-item for each polygon. Moreover,
we use a separate kernel to pre-compute the polygon projec-
tions that are stored in global memory so as to be used by
all work-groups working in parallel. Similarly to the PDA,
the index arrays are initialized with an unset value. For each
work-group, we update the index arrays (with the polygon
indices) when the polygons covered by a given work-item
project into the corresponding image cell. This process is illustrated in Figure 4 (b) where the colors in the output image encode the image cells.

**Optimization of Arrays of Polygon Indices** As illustrated in Figure 4 (c): the indices are compacted at the start of each cell row using an algorithm adapted from the parallel reduction of [Ope10]. Tests show that parallel reduction makes rendering 12.5 times faster than when using an algorithm that sequentially iterates through the indices and copies index values while tracking the last copied position. Note that we have also tested parallel reduction with the PDA and found that it did not produce any noticeable performance improvements; we believe that this is explained by the added access latency as the index arrays are stored in global memory.

![Cell Driven Approach](image)

**Figure 4:** Cell Driven Approach to ray-polygons depth map calculations. (a) The output image is segmented into cells shown as colored areas. Polygons are labeled with unique indices 1 to 4. (b) Index arrays of each work-item in local memory after the partitioning step. (c) Compacted index arrays in local memory.

### 4.3. Compositing

The compositing step is executed in a ray tracer that is easily parallelized as ray operations are highly independent. In our implementation we assign a work-item to each ray corresponding to each image pixel, e.g. a 512x512 image will have 512x512 work-items that can be organized into 64x64 work-groups if each work-group contains 8x8 work-items. The volume is loaded on GPU in the form of a 3-D image, called image object.

We define on each ray a far point and a near point, as illustrated in Figure 5, by the intersections of the ray with the bounding box of the volume. In our implementation we use the ray-box intersection algorithm [Gla90] to compute the near and far points of each ray. The renderer proceeds by moving along each ray and accumulating the color and opacity for the pixel where the ray originates. We use back to front alpha blending [Tho91] for compositing colors either of a voxel or of a polygon sample. The renderer executes the following steps according to where the current sampling point is located with respect to the far and near points:

1. Composite polygons farther than the far-point.
2. Composite polygons and voxels within the bounding box of the volume. Inside the volume, rays are traced along equally spaced intervals, called ray steps. In our implementation the step width is a parameter that can be adjusted automatically to render the smallest objects in the scene. At each rendering step we repeat the following steps:
   a. Sample and composite the volume. Sample the image object of the volume, resulting into a scalar value that is then mapped to RGBA by a transfer function. The transfer function is used to model the emission and absorption coefficients of the voxels. This RGBA value is then composited into the accumulated color and opacity of the pixel where the ray originates.
   b. Composite polygons. The color and opacity of intersecting polygons are composited into the accumulated color and opacity of the pixel where the ray originates
3. Composite polygons nearer than the near-point.

In the above steps 1, 2.b and 3, to find which polygons intersect a given ray, the renderer uses the depth arrays previously computed, and compares the depth values in the arrays with the distance from the camera to the current sample position.

![Ray Tracing Geometry](image)

**Figure 5:** Ray tracing geometry with volume bounding box, near and far points and ray steps.

### 5. Embedding view-dependent interactive text labels

As a particular example of embedding dynamic textured polygons we extend our renderer to display embedded text
labels that can be used to effectively convey contextual information and procedures.

Unlike the polygonal objects that we have considered so far, whose orientation remains constant with respect to the volume, we want our text labels to always face the viewer. In order to do that, we choose to use a dynamic rectangle as the text label container and then map the texture of the label’s content onto the rectangle. The rectangle has fixed center in the volume and aligned sides with the screen axes, and it also scales accordingly for the label’s geometry.

By processing labels separately from the rest of the polygons in the scene, we exploit their specific geometry to accelerate the depth map generation. Because in general only a small number of labels is present in the scene, we use CDA for best performance. Labels are processed differently in the following steps:

1. **Associate Text Rectangles with Image Cells**
   The coordinates of the text rectangle changes with the view, however, its projection in the screen space is bounded with the static rectangle whose center is the screen projected label position and whose size is the size of the rectangle in world space. If a ray does not intersect with the bounding rectangle, it does not intersect with the rectangle itself either, thus we take the static bounding rectangle for sorting the corresponding text rectangle into image cells.

2. **Ray-Rectangle Intersections for Generating Depth Arrays**
   For each text label, its container rectangle lies in a plane that is perpendicular to the vector from the camera to the label position in the volume. Any ray’s intersection distance to the plane can be easily calculated as illustrated in Figure 6. To decide whether the intersection point i is within the rectangle, simply by comparing $i_p_x$ and $i_p_y$ with `HALF_WIDTH` and `HALF_HEIGHT` respectively, as shown in Figure 7. Distances to valid intersections are sorted as the depth array for each cell.

3. **Compositing**
   The texture map is generated by rendering compacted textual contents of all labels into an off-screen buffer, and is then passed to GPU as a 2-D image object. As a result, each label has its own size and offset relative to the whole texture map. To composite the color of the text label for any intersection point i, we sample the texture map of label contents by using the absolute texture coordinates of i, which is calculated based on the relative coordinates as illustrated in Figure 7.

To add a pleasant click-n-highlight feature for text labels, we assign one bit for each label indicating whether it is selected or not. Initially the bit is set to unselect (0). When a mouse click event happens, in the step of depth arrays generation, we set the bit as select (1) for the nearest text plane hit by the ray originated from the mouse position. While in the rendering kernel, when compositing a text label with the selected bit, we blend its color with a brighter alpha value (as long as it’s not overflow). This simple technique enables text labels to be selected and highlighted without adding extra intersection or rendering time. It can also be applied to polygon selection if needed.

6. **Performance Analysis and Experimental Results**
   Our hybrid renderer was tested on a desktop (dual-core Intel®Xeon® 2.0GHz CPU, 4G RAM, NVIDIA®GeForce®260 GPU, Windows Vista) and on a netbook (Intel®Atom N450 1.66GHz CPU, 2G RAM, NVIDIA®ION GPU, Windows 7). All tests generate 512x512 output images.

   Because processing times are dominated by global memory read/write operations, we compute the time complexity based on global memory accesses. Given m polygons, n cells, p image pixels, u CUs and c as the capacity of the local memory, the time complexity of the PDA and the CDA
are $O\left(\frac{m_n^2}{n^2}\right)$ and $O\left(\frac{m_n^2}{np}\right)$ respectively. Under the testing condition, $n$, $p$, $u$ and $c$ are constant, $c \ll p$, we get two conclusions: 1) the performance of both approaches are linear in the number of polygons, and 2) CDA outperforms PDA in most cases. One should consider using PDA only when the geometry is too large to fit in local memory. The testing results on desktop as illustrated in Figure 8 can back up these conclusions.

Figure 8: Performance comparison between PDA and CDA obtained on desktop.

Figure 9 shows a scene of a volumetric Boston teapot (256x256x178 voxels) and a polygonal Utah teapot (896 polygons), full Phong shading is applied to the scene and different shading properties are set to volume and polygons. The CDA implementation can achieve 3fps on the netbook and 58fps on the desktop on average. The Utah teapot is opaque in the left image and translucent in the right for comparison. The lobster shape at the center of the volume of the Boston teapot is clearly visible through the translucent mesh of the Utah teapot. Figure 10 also shows how polygon transparency can be used to improve depth perception.

Figure 9: Polygonal Utah teapot (red) embedded in the volumetric Boston teapot (grey).

Figure 10: Skull with opaque and translucent cylinders.

Figure 11 compares images rendered with the SDM approach of [Vor] and with our MDM CDA. [Vor] can only embed one cylinder while the CDA supports more. Translucent polygons provide more details of the scene.

Figure 11: Walnut and semi-transparent cylinder(s). (a) and (d): SDM approach of [Vor]. (b), (c), (e) and (f): MDM CDA renderings. (c) and (f): Modulating the volume/polygon densities helps in visualizing the scene.

Figure 12 shows a scene of a textured Utah teapot and a Boston teapot.

Figure 12: Boston teapot and Utah teapot texture mapped with an opaque Lena image. Full Phong shading is apparent from the highlights.

Figure 13 shows different views of 9 Utah teapots embedded in a Boston teapot. Each Utah teapot is attached with a text label which orients with the view.

Figure 13: Differently colored and shaded teapots with text labels. Full Phong shading is applied to images. (b): label tp1 is selected.
7. Conclusion And Future Work

This paper proposes a novel MDM based approach for rendering polygons and volume that can meet the visualization requirements of scalable, interactive and improved depth perception. Specifically, it has the following advantages: 1) it enables polygon transparency in volume rendering which significantly improves the depth perception; 2) it is easy to apply different color, opacity, texture mappings and shading properties to both volume and each individual polygon; 3) it is easy to be extended to embed other types of objects and to add new features; 4) its implementation is platform-independent which can be deployed everywhere.

One common disadvantage poses on all GPU computing algorithms is that the dependence on the limited GPU memories could be a bottleneck. This can be alleviated to some extent by using smaller work-groups (local memory bottleneck) and multiple passes (global memory bottleneck). Specifically, to accommodate a large number of polygons, if the local memory is the bottleneck, we can further split polygons on z axis based on the current 2-dimensional partition so that the size of local memory is enough for smaller work-groups; if the global memory is the bottleneck, we can process and embed polygons in multiple rendering passes so that the size of global memory is enough for each pass. Another disadvantage that bugs all ray tracing renderers is the artifact introduced by the nature of ray tracing. This can be greatly alleviated by several well-developed techniques like selective super-sampling.

There are a lot of future work to do in order to make the renderer better for practical use, to list a few: 1) apply traditional optimization techniques (Early Ray Termination, Empty Space Skipping, etc.) in volume rendering; 2) apply best practices for OpenCL and GPU computing; 3) apply techniques such as selective super-sampling to alleviate ray tracing artifacts; 4) extend it to embed other types of objects; 5) add more features like global illumination, shadows, etc. to better visualize the volume.

8. Acknowledgements

The volume data used for the experimental results was downloaded from http://www.volvis.org/. OpenCL is a trademark of Apple Inc., used under license by Khronos.

References


[Voe] VOREN, Tech. rep., Visualization and Computer Graphics Research Group, University of Münster. 1, 2, 7
