Proceedings of the IASTED International Conference
Computer Graphics and Imaging (CGIM 2013)
February 12 - 14, 2013 Innsbruck, Austria

A COMPARISON OF GPU BOX-PLANE INTERSECTION ALGORITHMS FOR DIRECT VOLUME RENDERING

Stefan Zellmann
Chair of Computer Science
University of Cologne
Cologne, Germany
email: zellmans@uni-koeln.de

Ulrich Lang
Chair of Computer Science
University of Cologne
Cologne, Germany
email: lang@uni-koeln.de

ABSTRACT

In order to avoid load imbalances on the GPU during direct volume rendering, a common scheme was to move the generation of proxy geometry, that quite often consists of polygons retrieved through box-plane intersections, from the CPU to the vertex stage of the GPU. Nowadays, with the unified shader architectures implemented by modern graphics hardware, usually non of the programmable stages will starve anymore. Nevertheless, redistributing calculations for proxy geometry generation from the vertex stage to the newly introduced geometry stage of the graphics hardware results in some serial computations that can be performed more finely grained, thus hinting the graphics driver to schedule more tasks in parallel. We propose different implementations of box-plane intersection algorithms on the GPU that use the vertex- as well as the geometry stage and compare their performance on modern graphics hardware.

KEY WORDS

Volume Rendering, Rendering Algorithms and Systems, Box-Plane Intersection Algorithms, Geometry Programs

1 Introduction

Visualization of large volumetric datasets is an interdisciplinary requirement. Meteorologic simulations often result in huge datasets with many time steps. CT- or MRT scans produce large amounts of data defined on regular grids that need to be visualized, while flow simulation results in high-resolution data with a need for high-quality visualization methods. Visualization algorithms need to keep pace with the ever-growing sophistication of measuring devices generating volume datasets that grow with the 3rd power of their spatial resolution. High screen resolutions supported by newer displays produce an even higher impact on the usually fill rate bound volume rendering algorithms.

One possible categorization differentiates between image-order and object-order algorithms for rendering of volume datasets. With image-order algorithms like ray-casting, sampling is performed per pixel, while object-order approaches sample the volume using some kind of proxy geometry that is directly rendered to the framebuffer by e. g. using rasterization. With modern GPUs, trilinear interpolation is a cheap operation, which makes GPUs especially suitable for rendering of volume datasets defined on regular, three-dimensional grids. Object order algorithms for GPUs usually use a planar proxy geometry that is parallel to the viewing plane slicing a 3D texture, while in the image-order case ray-casting could e. g. be implemented in a shader [1]. In both cases, with the fixed-function graphics pipeline model implemented by legacy hardware, the workload distributed to the fragment processing stage would be inefficiently high compared to the workload for the vertex stage. Rezk Salama and Kolb [2] counter this load imbalance by moving the generation of proxy geometry, which intuitively seems to be a task that is hard to parallelize, to the vertex processing stage on the GPU. Modern GPUs typically do not implement the stages of the fixed-function pipeline anymore. Rather than that, they nowadays usually provide processing units that are capable of performing more general computations [3]. Unified shader units can be used for vertex processing as well as fragment processing, so that typically non of both stages gets starved anymore, since a dynamic amount of processing units can be used for the task with the higher workload. With graphics APIs exposing even more stages of the fixed-function pipeline to be programmed in shaders, it is possible to distribute the workload more evenly, hinting the GPU to benefit from parallelization to a higher degree.

Proxy geometry generation for volume rendering with 3D texture slicing becomes a bottleneck if either the amount of sampling planes is unrealistically high, or if the volume is not represented through one single box, but through many boxes, as it is e. g. necessary for empty-space skipping [4]. In that case, the volume is usually still stored in a single 3D texture uploaded to the GPU, but a data structure like a more coarsely grained grid or an octree is placed on top of the volume dataset. This way, in a preprocessing step whole regions can be identified that do not contain data relevant for rendering and can thus be skipped. Efficient empty-space skipping may require many boxes, resulting in an extensive workload due to box-plane intersection calculations necessary to build up the proxy geometry. The authors from [2] are able to significantly speed up the volume rendering task by parallelizing the proxy geometry generation and moving that step to the vertex processing stage of the GPU. We explore variants of their box-plane intersec-

DOI: 10.2316/P.2013.797-019

tion algorithm that use the newly added geometry program features to hint modern GPUs to expose a higher degree of parallelism and compare our variants to the speedup gained by the original implementation.

Our paper is organized as follows. In section 2, we review current and more recent research that relates to our work. In section 3, we motivate the box-plane intersection algorithm from [2] our work is based upon. Section 4 presents variants of this algorithm using geometry programs exposed by modern graphics hardware, while section 5 compares our implementations to the reference implementation using performance measurements. The last section 6 concludes this publication.

2 Related Work

Westermann and Ertl [5] explain how 3D textures can be exploited to sample volume data using trilinear interpolation. Similar to Dachille et al. [6], the authors use the 3D texture capabilities to implement object-order direct volume rendering. The authors furthermore investigate how to use the 3D hardware to explore the volume using planar as well as arbitrary geometries to clip away parts of the volume. The authors from [7] mention that spherical shells can be incorporated to sample the 3D texture so that step sizes are approximately constant, as are those obtained by ray-casting. However, the authors find this approach impractical due to the huge amount of geometry that needs to be generated, transferred to the GPU and rendered. On top of this, the authors outline how empty-space skipping can be benefitial to accelerate volume rendering, which in their case is applied to volume ray-casting. By applying an additional data structure on top of the data structure containing the volume data, the sampling distance along a ray can be increased when entering empty volume regions. They state that such a data structure can e. g. be an octree hierarchy. Data structures like grids and octrees usually subdivide the volume dataset into equally sized grid cells, resulting in rather inflexible subdivision schemes that are not able to adapt to non-uniformly sized empty regions throughout the volume. Li et al. [4] counter this problem by organizing the volume using BSP trees that store boxes of dynamic size at their nodes. Rezk Salama and Kolb [2] describe how a planar proxy geometry for direct volume rendering can be generated efficiently on the GPU. They rely on efficient proxy geometry generation in order to perform empty-space skipping. The algorithm they propose is covered in detail in the following section 3.

Decaudin and Neyret [8] apply geometry programs to volume rendering. They propose to extend the concept of billboards to so called volumetric billboards, where instead of using a flat representation as a coarse approximation for far away 3D objects in a scene an also coarse, but volumetric representation is used. They generate the volumetric representation on the fly from actual 3D objects and organize them using MIP-maps. They also propose to use geometry programs for intersection calculations, but in contrast to our approach, they use a planar proxy geometry to sample triangular prism-shaped cells. Other applications of geometry programs for visualization can be found in [9] and [10], where geometry programs are used for dynamic mesh refinement and primitive replication for piecewise perspective projections.

3 Vertex Program for Box-Plane Intersection

With object-order algorithms for volume rendering utilizing the 3D texture capabilities of modern GPUs, volume datasets are nowadays usually sampled using a planar proxy geometry. Viewport aligned slices are drawn in z-order and the GPU interpolates the 3D texture at the respective sampling positions on the fragment processing stage. The GPU is usually configured to use alpha blending using e. g. the *over*-operator [11], so that alpha compositing is performed for the trilinearly interpolated samples from the fragment stage. With this approach, viewport aligned slices need to be recalculated on the fly each time the camera is moved. Rezk Salama and Kolb [2] propose a highly optimized algorithm to calculate the polygon resulting from the box-plane intersection in parallel using a vertex program.

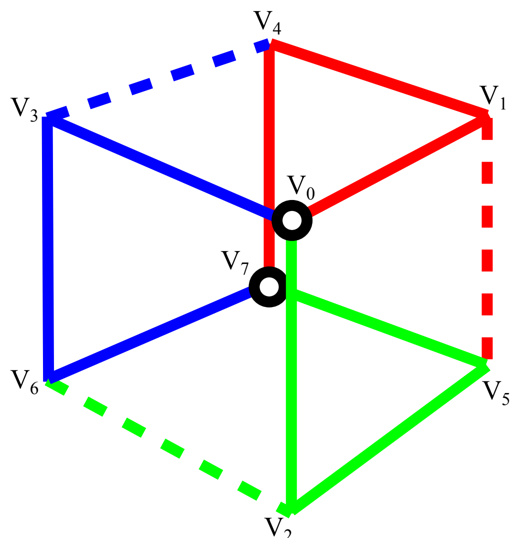

Figure 1. Three unique paths from V_0 to V_7. The intersection points with the dotted lines must be filled in between the intersection points with the solid edges along the respective paths.

First of all, they identify V_0 - the vertex of the box that is closest to the camera - as well as V_7 - the vertex that belongs to the opposite corner. Then they identify three shortest paths from V_0 to V_7, that share no edges at all (cf. Figure 1). By determining the vectors corresponding to the three edges that make up each path and forcing the vectors to form a left handed coordinate system per path, such paths can be defined uniquely. A viewport aligned plane

that intersects the box must have exactly one intersection per path. Box-plane intersections result in polygons having three to six vertices. I. e. if the polygon resulting from the box-plane intersection happens to be a triangle, this is found by intersecting the plane with the three independent paths, resulting in intersection positions P_0, P_2 and P_4. In order to find the up to three additional vertices, one intersects the plane with the three dotted eges from Figure 1 and places the resulting vertices P_1, P_3 and P_5 just in between the three vertices that are already calculated. If no intersection is encountered, the previous vertex is duplicated, resulting in the following initialization scheme:

P_0 = Intersection with $E_{0\to1}$, $E_{1\to4}$ or $E_{4\to7}$

P_1 = Intersection with $E_{1\to5}$, otherwise P_0

P_2 = Intersection with $E_{0\to2}$, $E_{2\to5}$ or $E_{5\to7}$

P_3 = Intersection with $E_{2\to6}$, otherwise P_2

P_4 = Intersection with $E_{0\to3}$, $E_{3\to6}$ or $E_{6\to7}$

P_5 = Intersection with $E_{3\to4}$, otherwise P_4

This lends itself well to an implementation using a vertex program. With these, no new vertices can be generated, so that the program must be passed six locations that it moves to appropriate intersection positions. By moving two vertices to exactly the same location, the GPU will cancel out the respective edge. That way, one can generate polygons with three to six corners. Finding the intersections is simply a matter of substituting the plane equation into the edge equation and solving for the intersection position. P_0, P_2 and P_4 will necessarily be valid intersections, while P_1, P_3 and P_5 are optional. Parallelism is achieved on a per-vertex level. To avoid branching that is expensive on GPUs, each vertex program performs four edge-plane intersection tests in a loop. The programs responsible to find the intersections with the even index process all edges on their path and one invalid edge in addition during the last loop iteration. If the first intersection is found, the loop exits. Since the intersections with even index necessarily exist, the loop will never reach the fourth iteration anyway. After finding the intersection, the vertex is moved to the intersection position and texture coordinates to perform post classification on the fragment stage are adjusted accordingly. The vertex program responsible for the intersections with odd index intersects the same path with the same plane. I. e. during iterations one, two or three, the intersection found by the corresponding program processing the vertex with the even index will be found. During the fourth iteration, the edge that distinguishes this path from the other one will be processed. If no intersection is found, the vertices with the odd and the even index will have the same location. If an intersection is found, the polygon resulting from the box-plane intersection contains that corner. Edges are passed to the vertex programs through lookup tables, as are the eight vertices of the box.

4 GPU Variants of the Intersection Algorithm

Rezk Salama and Kolb argue that calculations can be moved from the CPU to the vertex stage of the GPU. This results in better load balancing on the GPU, if there is a dedicated number of processors responsible for the vertex stage and for the fragment stage, respectively. With unified shader architectures in newer GPUs, a varying number of general purpose processors can be assigned to both stages of the graphics pipeline. To better accomodate unified shader architectures, we propose to subdivide the intersection routine described above into more finely grained subroutines, hinting the graphics driver to schedule them with a higher degree of parallelism. To accomplish this, we exploit the newly added geometry stage of the graphics pipeline.

We propose three novel variants of the algorithm described above. The first variant implements slight modifications to the reference implementation while keeping all calculations in a vertex program exclusively. The two remaining variants utilize the geometry shader stage for proxy geometry generation to varying degrees. All variants were integrated into the direct volume rendering library Virvo [12].

4.1 Variant One - Slight Modifications to the Reference Implementation

Our first variant mostly reimplements the reference implementation from Rezk Salama and Kolb. Our aim is to reduce the memory footprint of their original vertex program.

During each loop iteration in the original vertex program, an edge-plane intersection test is performed. To identify the edge to process during each iteration, a rather complicated scheme of lookups into fixed size arrays is used to determine the two vertices that make up the edge. First the index $[0..7]$ of the box vertex closest to the viewer is identified. Using this index and multiplying it by 8, a lookup is performed into the *sequence* array from Listing 1. If e. g. the index of the closest vertex is 3, the vertex program will process the box vertices in the sequence $3, 6, 5, 0, 7, 2, 1, 4$. Depending on the intersection position P_0 through P_5 to process, the correct paths (cf. Figure 1) are identified by looking up the first vertex of an edge from $v1$ and the second vertex from $v2$. Note that some of the array entries have a negative index. These are the invalid edges that will never be processed. With the correct indices for the paths, the object space coordinates are looked up in the $vertices$ array. All the arrays are passed to the vertex program using uniform variables, which substantially increases the memory usage of one program instance.

We propose to perform most of the lookups on the CPU. That way, in addition to the reduced memory usage in the vertex program, lookups only have to be performed once per box, instead of once per vertex. The $sequence$ array is never passed to the vertex program. Instead, the

```
int sequence[64] = { 0, 1, 2, 3, 4, 5, 6, 7,
                     1, 2, 3, 0, 7, 4, 5, 6,
                     2, 7, 6, 3, 4, 1, 0, 5,
                     3, 6, 5, 0, 7, 2, 1, 4,
                     4, 5, 6, 7, 0, 1, 2, 3,
                     5, 0, 3, 6, 1, 4, 7, 2,
                     6, 7, 4, 5, 2, 3, 0, 1,
                     7, 6, 3, 2, 5, 4, 1, 0 };

// edge vertex one
int v1[24] = { 0, 1, 4, -1,
               1, 0, 1, 4,
               0, 2, 5, -1,
               2, 0, 2, 5,
               0, 3, 6, -1,
               3, 0, 3, 6 };

// edge vertex two
int v2[24] = { 1, 4, 7, -1,
               5, 1, 4, 7,
               2, 5, 7, -1,
               6, 2, 5, 7,
               3, 6, 7, -1,
               4, 3, 6, 7 };

// object space vertices of the box
vec3 vertices[8] = { ... };
```

Listing 1. Uniform lookup tables passed to the vertex program from the reference implementation. We propose to move the lookup scheme to reconstruct paths from the GPU vertex stage to the CPU to reduce the memory footprint of the vertex program.

$vertices$ array is set up in the correct sequence on the CPU already, i. e. before passing it to the vertex program.

We on top of this propose to traverse slightly different paths to find the intersection positions $P_0 - P_5$:

P_0 = Intersection with $E_{0\to1}$, $E_{1\to4}$ or $E_{4\to7}$

P_1 = Intersection with $E_{0\to1}$, $E_{1\to5}$ or $E_{5\to7}$

P_2 = Intersection with $E_{0\to2}$, $E_{2\to5}$ or $E_{5\to7}$

P_3 = Intersection with $E_{0\to2}$, $E_{2\to6}$ or $E_{6\to7}$

P_4 = Intersection with $E_{0\to3}$, $E_{3\to6}$ or $E_{6\to7}$

P_5 = Intersection with $E_{0\to3}$, $E_{3\to4}$ or $E_{4\to7}$

The paths with an even index form a left handed coordinate system, while the corresponding paths used to obtain the intersection position with an odd index form a right handed coordinate system. Since the first edge of two corresponding paths is identical, the correct order and distinctness of the intersection positions is maintained.

The new traversal scheme enables us to store the paths as a consecutive array of vertices and thus to skip one of the arrays $v1$ and $v2$ in favor of one array v of size 24 (cf. listing 2). As a result of this, only three rather than four edge-plane intersection tests are necessary per vertex.

```
int v[24] = { 0, 1, 4, 7,
              0, 1, 5, 7,
              0, 2, 5, 7,
              0, 2, 6, 7,
              0, 3, 6, 7,
              0, 3, 4, 7 };
```

Listing 2. Only one uniform lookup table is necessary for our revised vertex program. This substantially reduces the memory footprint of one vertex program instance compared to the reference implementation.

4.2 Variant Two - Proxy Geometry Generation using a Single Geometry Program

Our second variant uses the geometry stage of the rendering pipeline exclusively. With the OpenGL shading language (GLSL) we used for our implementation, geometry programs accept one or more vertices from the preceding vertex stage and are able to emit zero to many vertices. The input primitives are limited to a subset of all available OpenGL primitives, as are the output primitives that can be emitted by a geometry program. In our case, the presence of a *polygon* 3D primitive would have been convenient for output, since the reference vertex program relies on the distinct order in which the polygon vertices are emitted. The same order could have been maintained if a *triangle fan* primitive would have been available, which is not the case either, so that we have to make do with the *triangle strip* 3D primitive as output of our geometry program.

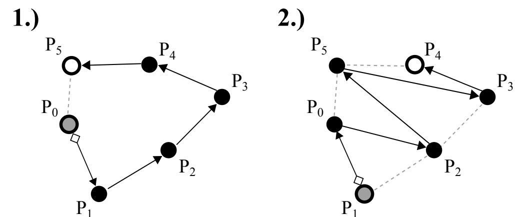

Figure 2. The right image shows how to create a triangle strip equivalent to the polygon depicted in the left image. In order to be able to generate each kind of primitive from triangles to hexagons by omitting selected vertices, the strip must start with P_1.

Our second variant of the reference GPU implementation moves the complete workload of variant one to a geometry program. Input to the geometry program is a single point primitive. Output is a triangle strip consisting of up to four triangles, i. e. one intersection polygon is created from one single vertex. This implementation strongly resembles the implementation described in section 4.1. This time, a vertex program is only needed to pass through the single vertex to the geometry program. In the geometry program, a loop iterates over all six possible intersection positions.

Exactly as it is the case in the reference implementation, intersections that do not exist are omitted by emitting duplicate vertices. In order to generate triangle strips, intersection positions need to be evaluated in a different order than this is true for the polygon primitive. Figure 2 illustrates the exact traversal to create a triangle strip equivalent to the polygon output by the implementation from section 4.1, guaranteeing that each pair of corresponding intersection positions is processed consecutively.

Compared to the implementation using a single vertex program, this implementation trades parallelism - here vertices are processed in sequence, while six vertex programs can be scheduled in parallel - for bandwidth, since only the sixth part of the original vertex load needs to be transferred from the CPU to the GPU.

4.3 Variant Three - Distribute Workload Among a Vertex Program and a Geometry Program

Our third variant distributes the workload for proxy geometry generation more evenly among the vertex- and the geometry stage. This time, we choose triangles as input primitives to the vertex program. In the vertex program, we perform the three edge-plane intersections for which we know that they exist, while the geometry program processes the three optional intersection positions.

```
 if (intersect(P1) emit(P1))
emit(P0) // passed through from vertex shader
emit(P2) // passed through from vertex shader
if (intersect(P5))
{
  emit(P5)
  if (intersect(P3) emit(P3))
  emit(P4) // passed through from vertex shader
}
else
{
  emit(P4) // passed through from vertex shader
  if (intersect(P3) emit(P3))
}
```

Listing 3. Pseudo code to emit vertices from a geometry shader so that triangle strips are generated from input triangles. Branching depends on the existence of P_5.

Since we have to cope with triangle strips as the only feasible output primitive, branching in the geometry program is unavoidable. Taking all combinations of possible intersections $P_0 - P_5$ and given that P_0, P_2 and P_4 necessarily exist, only one combination may result in a triangle or a hexagon, while there are three distinct combinations that can lead to quadrangles or pentagons, respectively. Generating triangle strips as illustrated in Figure 3 for each combination results in a generation scheme that minimizes branching. Listing 3 outlines this generation scheme. Different branches have to be taken depending on the existence of intersection position P_5.

Splitting the intersection computations to be performed by a vertex program and a geometry program in conjunction seems to be the most promising approach regarding performance. Primitive emission poses a natural barrier. With the two variants using a vertex program or a geometry program solely, only one such barrier exists. With the combined approach, primitives are emitted by both the vertex program and the geometry program, while less intersection positions have to be processed up to each barrier, resulting in a finer granularity and thus in a higher potential for the graphics driver to benefit from a higher degree of parallelism. Additionally, only half the geometry needs to be transferred from CPU main memory to the video memory of the graphics card compared to the mere vertex program solution.

5 Results

We evaluate the performance of the three GPU box-plane intersection algorithm variants using three different hardware setups:

- Workstation: this setup consists of a graphics workstation equipped with an 8 Core Intel Xeon X5472 CPU with 3.00GHz and an NVIDIA Quadro FX5800 graphics card from the NVIDIA professional segment, that comes with 4GB GDDR3 video memory.
- Desktop: with this setup, a 4 Core Intel Xeon 5160 CPU with 3.00GHz and an NVIDIA GeForce Series GTX480 graphics card from the NVIDIA consumer series are built into a standard tower casing. The graphics board is equipped with 1.5GB GDDR5 video memory.
- Notebook: for this setup, we use an Apple MacBook Pro from the "Early 2011" generation, that is equipped with an Intel Core i7 CPU with 2.00GHz as well as an AMD Radeon HD 6490M graphics card with 256MB GDDR5 memory.

The operating system for the *Workstation* and *Desktop* setups is a Linux distribution. With the *Notebook* setup, we use Mac OS X 10.7 (Lion). Performance is measured as follows for all hardware setups: With the Linux setups, that run an X server, the *composite* extension is deactivated, that is responsible for desktop effects and would possibly compete for GPU resources. We found no similar means to deactivate desktop effects under Mac OS X and have to cope with the default configuration in the *Notebook* case.

For our tests, a volume file with a resolution of $256 \times 256 \times 225$ and one byte of data per voxel is loaded into our volume rendering application and rendering is performed using a window with a screen resolution of 512×512 pixels. The volume utilizes a regular grid for empty-space skipping, although due to the transfer function we use, there are no empty regions and all grid cells need to be evaluated (cf. Figure 4). Each cell covers $64 \times 64 \times 64$

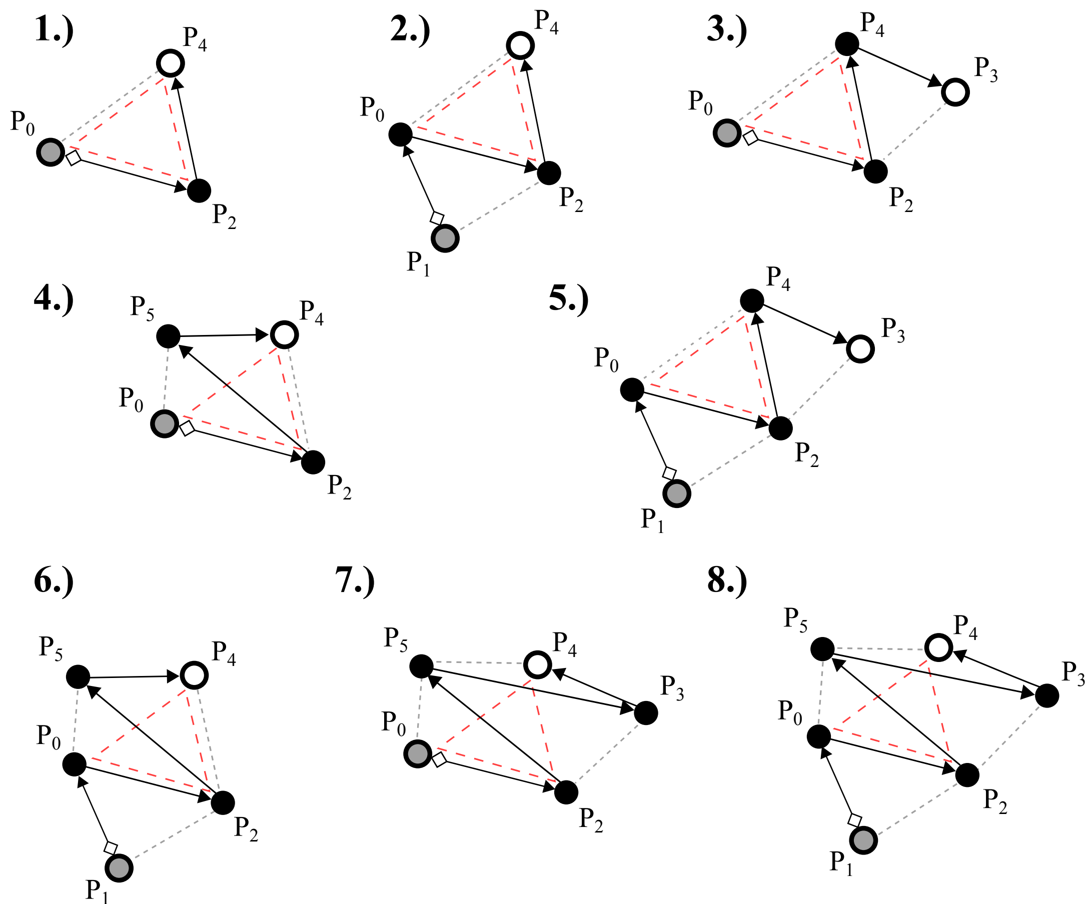

Figure 3. Eight ways to fill in intersection positions P_1, P_3 and/or P_5 generated by the geometry program in between intersection positions P_0, P_2 and P_4. To generate *triangle strip*s, the optional corners must be filled in at appropriate positions. The first vertex emitted by the geometry program has a gray filling, while the last vertex has a white filling. The black, solid arrows depict the order in which the corners are emitted, while the gray, dotted edges are only there to indicate the convex hull of the polygon. The red, dotted edges depict the outline of the triangle spanned by P_0, P_2 and P_4 originating from the vertex program. Case 1.) shows the trivial triangle strip for three corners. Cases 2.) through 4.) show how to generate triangle strips for the three possible quadrangles. Cases 5.) through 7.) show the same for the pentagons, while case 8.) shows how to generate a triangle strip for the hexagon.

voxels. Because we also use the regular grids to manage volumes that do not fit completely into video memory, one additional voxel needs to be stored at the cell borders for trilinear interpolation, resulting in the volume being organized into $5 \times 5 \times 4 = 100$ grid cells (cf. Figure 5). The volume is sampled using 426 planes. At the beginning of one test run, the volume is moved to the world coordinate origin and no further transformations are applied to it. The camera is translated so that the volume is fully visible all the time. Then, the volume is rotated 90 times in steps of $2°$ along the x-axis, resulting in an overall rotation of $180°$. This procedure is repeated for the y-axis and the z-axis, respectively. Each time the camera is moved, the time needed to render one image is measured. At the beginning of one test run, we perform this procedure once but disregard the results. We follow this approach since we registered an increase in performance after some time of interaction with the volume and believe that this is due to the fact that when rendering the first few frames, the texture caches of the GPU are not sufficiently filled with data yet. This behavior was most significant when rendering with the Quadro FX5800 GPU. After filling the texture caches, we perform the three rotations 10 more times, resulting in $2,700$ frames rendered during one test run, for which we calculate the average and standard deviation. We found that clearing video memory and loading up the 3D texture to the GPU again resulted in slightly different rendering times compared to the rendering times measured before. We believe this to be another side effect of the caching implementation of the graphics driver and account for it by performing each test run - i. e. performing the three rotations 11 times and dropping the results of the first three rotations - 10 times, each time reloading the 3D texture to video memory anew, and finally averaging over these 10 runs. On average, $6,444$

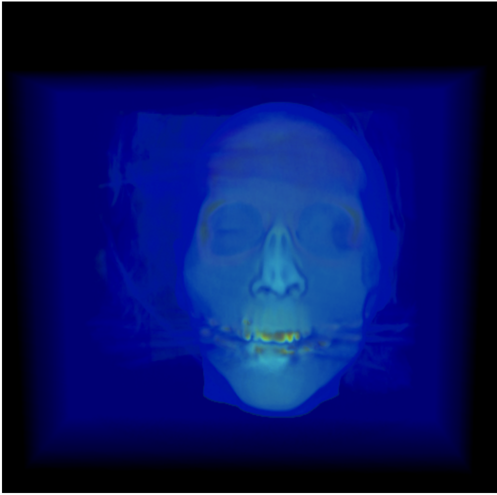

Figure 4. The transfer function we apply for our tests maps all voxels to nontransparent colors.

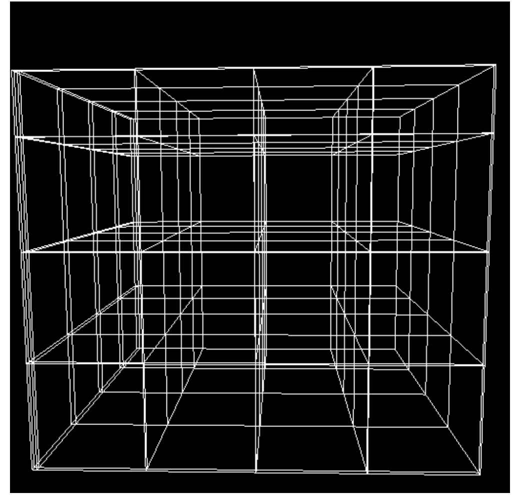

Figure 5. The regular grid superimposed on top of the volume from Figure 4.

box-plane intersections are calculated per frame. When rotating the volume three times around each primary axis, the overall number of box-plane intersections amounts to $1,739,290$.

Volume rendering is performed using the *over*-operator for alpha-blending, which results in a high workload on the fragment stage of the rendering pipeline of the GPU. Because we are mostly interested in the performance of the intersection calculations that compete with the fragment programs for unified shader resources, we propose a second test setup in addition to the ordinary alpha blending test setup. With this second test setup, we still instruct the GPU to generate polygons or triangle strips as output primitives. But by setting the *polygon mode* to *POINT* through the graphics API, we output only the corners of the 3D primitives to the fragment program, resulting in a negligible amount of work for that stage of the rendering pipeline. In the following, we refer to these two modes by the terms *Blending* and *Points*.

Additionally, we apply this performance measuring routine to a single threaded CPU implementation as well as an exact reimplementation of the algorithm presented by Rezk Salama and Kolb for comparison. Table 1 summarizes the results for the *Blending* mode, while Table 2 outlines the same performance measurements for the *Points* mode.

On the *Workstation* setup, we find the variants using geometry programs to outperform the variant using a vertex program by appr. 10%. We find it noteworthy that with this setup, rendering times are virtually the same for both the *Blending* and the *Points* mode. The post classification fragment program we apply for transfer function evaluation performs merely three tasks: first a lookup into the 3D texture with the volume data. Then a trilinear interpolation and with the outcome a lookup to retrieve color and opacity from a 1D texture. Given the large amount of video memory the NVIDIA Quadro graphics card is equipped with, we surmise that both textures can be completely kept in caches and lookups are cheap. Since trilinear interpolation is a cheap operation for GPUs anyway, the workload for the fragment stage is neglible whether or not only the vertices of the polygons and triangle strips, or their filled and translucent counterparts are drawn. With the *Desktop* setup, we find the vertex program variant superior to both variants utilizing geometry programs. With this test setup, the workload for the fragment stage seems substantially higher, since rendering with alpha blending results in much higher rendering times in all cases. In both the *Workstation* as well as the *Desktop* case, using the GPU for proxy geometry generation is far superior to using the CPU.

Var.	Workstation	Desktop	Notebook
One	0.428 (0.003)	0.574 (0.002)	9.155 (0.026)
Two	0.394 (0.002)	0.594 (0.002)	13.21 (0.089)
Three	0.385 (0.002)	0.580 (0.002)	5.945 (0.049)
CPU	7.508 (0.007)	7.336 (0.015)	6.136 (0.086)
Ref.	0.428 (0.003)	0.577 (0.002)	9.972 (0.052)

Table 1. Results of our performance tests for the Blending mode, i. e. ordinary direct volume rendering using alpha compositing. Values depict the time (sec.) it takes to perform three $180°$ rotations around each primary axes in steps of $2°$ - resulting in 270 frames. This procedure is carried out 10×10 times and results are averaged. Values in parentheses display the standard deviation for the test runs. For comparison, we provide test results for a single threaded CPU implementation and for the reference implementation.

With the *Notebook* setup, we find that using shader programs in general and especially geometry programs are not able to outperform the CPU that significantly as it is the

Var.	Workstation	Desktop	Notebook
One	0.428 (0.002)	0.393 (0.002)	5.520 (0.027)
Two	0.386 (0.002)	0.410 (0.001)	13.99 (0.155)
Three	0.384 (0.002)	0.400 (0.001)	6.186 (0.052)
CPU	7.504 (0.016)	7.326 (0.015)	6.040 (0.034)
Ref.	0.431 (0.003)	0.400 (0.001)	6.309 (0.052)

Table 2. Results of our performance measurements for the Points mode. Here, rather than the polygons, only the vertices resulting from the intersection calculations are drawn by setting the polygon mode to POINT through the graphics API, resulting in a neglible workload on the fragment stage.

case for the other two test setups. Using the combined vertex / geometry program variant is slightly faster than using the single threaded CPU variant, while the mere geometry program variant even needs double the time of the CPU. With geometry programs, we encountered an interesting phenomenon: rendering with only the vertices from the intersections visible even resulted in a performance drop compared to rendering with alpha blending. We also note an increased deviation in rendering times compared to using only a single vertex program for intersection calculations. We can only speculate that shader programs are rather poorly supported by the Apple AMD drivers, though with ordinary volume rendering, the combined vertex / geometry program variant is slightly superior even to the CPU implementation.

6 Conclusion

We presented three variants of implementing box-plane intersection tests for direct volume rendering on the GPU using the programmable stages exposed by the OpenGL graphics API. For test setups with high performance graphics hardware, the GPU implementations outperform intersection tests on the CPU, just as Rezk Salama and Kolb proved in their original paper. Assigning some of the calculations to geometry programs can even further enhance performance slightly in some cases. Because geometry program support is poorly implemented on some hardware platforms, implementing the intersection computation in a vertex program will result in a more general solution and thus is preferable at this time. We presented a way to subdivide the proxy geometry generation so that both vertex stage and geometry stage can handle a compute intensive share of that task in parallel. Although this subdivision scheme results only in a slight performance enhancement on current hardware platforms, we believe it to be a natural enhancement to the original implementation by Rezk Salama and Kolb that can potentially outperform the mere vertex program implementation on future architectures.

References

[1] M. Hadwiger, J. M. Kniss, C. Rezk Salama, D. Weiskopf, & K. Engel, *Real-time Volume Graphics*, Natick, MA, USA: A. K. Peters, Ltd., 2006, ISBN 1568812663.

[2] C. Rezk Salama & A. Kolb, A vertex program for efficient box-plane intersection, *Proc. Vision, Modeling and Visualization (VMV)*, 2005, pages 115–122.

[3] E. Lindholm, J. Nickolls, S. Oberman, & J. Montrym, NVIDIA Tesla: A unified graphics and computing architecture, *Micro, IEEE*, 2008, *28*(2), 39–55.

[4] W. Li, K. Mueller, & A. Kaufman, Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering, *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03*, IEEE Computer Society, 2003 .

[5] R. Westermann & T. Ertl, Efficiently using graphics hardware in volume rendering applications, *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Request Permissions, 1998 .

[6] F. Dachille, K. Kreeger, B. Chen, I. Bitter, & A. Kaufman, High-quality volume rendering using texture mapping hardware, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 1998, pages 69–ff.

[7] J. Kruger & R. Westermann, Acceleration Techniques for GPU-based Volume Rendering, *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03*, IEEE Computer Society, 2003 .

[8] P. Decaudin & F. Neyret, Volumetric billboards, *Computer Graphics Forum*, 2009, *28*(8), 2079–2089.

[9] H. Lorenz & J. Döllner, Dynamic mesh refinement on GPU using geometry shaders, *Proceedings of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2008, pages 97–104.

[10] H. Lorenz & J. Döllner, Real-time piecewise perspective projections, *GRAPP 2009-International Conference on Computer Graphics Theory and Applications*, 2009, pages 147–155.

[11] T. Porter & T. Duff, Compositing digital images, *ACM Siggraph Computer Graphics*, 1984, *18*(3), 253–259.

[12] J. Schulze, U. Woessner, S. Walz, & U. Lang, Volume rendering in a virtual environment, *Immersive Projection Technology and Virtual Environments 2001: proceedings of the Eurographics Workshop in Stuttgart, Germany, May 16-18, 2001*, 2001, page 187.