

# Volume Rendering on Mobile Devices

Mika Pesonen

University of Tampere  
School of Information Sciences  
Computer Science  
M.Sc. Thesis  
Supervisor: Martti Juhola  
June 2015

University of Tampere  
School of Information Sciences  
Computer Science  
Mika Pesonen: Volume Rendering on Mobile Devices  
M.Sc. Thesis, 59 pages  
June 2015

---

The amount of GPU processing power for the mobile devices is increasing at a rapid pace. The leading high-end mobile GPUs offer similar performance found in medium category desktop PCs and laptops. Increased performance allows new and advanced computer graphics algorithms to be run on a mobile devices.

Volume rendering needs complex computer graphics algorithm that requires high processing power and memory bandwidth from the GPUs. In theoretical part of this thesis the theory behind volume rendering algorithms and optimization techniques are introduced. Additionally, results from the existing research for mobile volume renderers and common volume rendering algorithm selections are analyzed.

The study of this thesis focuses on different volume rendering algorithms that are evaluated and implemented on multiple different mobile GPU architectures. An analysis for the study results and suggestions for suitable mobile device volume rendering algorithms are provided.

Keywords: volume rendering, mobile devices, graphics performance, cross-platform development, OpenGL, WebGL

## Contents

1.	Introduction .....	1
2.	Volume Rendering .....	2
2.1.	Indirect volume rendering .....	2
2.1.1.	Contour tracing .....	2
2.1.2.	Cuberille .....	3
2.1.3.	Marching cubes .....	4
2.1.4.	Marching tetrahedra .....	5
2.2.	Direct volume rendering .....	6
2.2.1.	Shear-warp factorization .....	6
2.2.2.	Splatting .....	7
2.2.3.	Texture slicing .....	8
2.2.4.	Ray casting .....	9
2.3.	Classification .....	10
2.4.	Composition .....	11
3.	Local illumination models .....	13
3.1.	Gradient .....	13
3.2.	Phong .....	14
3.3.	Blinn-Phong .....	16
4.	Global illumination models .....	17
4.1.	Shadow rays .....	17
4.2.	Shadow mapping .....	17
4.3.	Half-angle slicing .....	18
5.	Optimization techniques .....	19
5.1.	Empty space skipping .....	19
5.2.	Early ray termination .....	20
5.3.	Adaptive refinement .....	21
5.4.	Bricking .....	21
6.	Existing solutions .....	22
6.1.	Interactive Volume Rendering on Mobile Devices .....	22
6.2.	Practical Volume Rendering in Mobile Devices .....	22
6.3.	Volume Rendering Strategies on Mobile Devices .....	23
6.4.	Interactive 3D Image Processing System for iPad .....	25
6.5.	Interactive visualization of volumetric data with webgl in real-time .....	25
6.6.	High-performance volume rendering on the ubiquitous webgl platform .....	27
6.7.	Visualization of very large 3D volumes on mobile devices and WebGL .....	28
6.8.	Research summary .....	29
7.	Evaluation .....	31
7.1.	GPU hardware specifications .....	33

7.2. Test setup .....	35
7.3. Slicing vs. ray casting .....	36
7.4. Local illumination .....	40
7.5. Optimization techniques .....	42
7.6. Native runtime vs. web runtime .....	45
8. Conclusion .....	47
References .....	48
Glossary .....	51
Acronyms .....	52
Appendices .....	53
Slicing shader program .....	53
Ray casting shader program .....	55

## 1. Introduction

There has been rapid advances in the mobile graphics GPUs in the recent years. Performance and programmability of mobile GPUs are reaching laptop levels. A popular graphics performance program called GFXBench (1) can be used to compare mobile devices to laptops. Results for some of the devices benchmarked by GFXBench are illustrated in Figure 1.

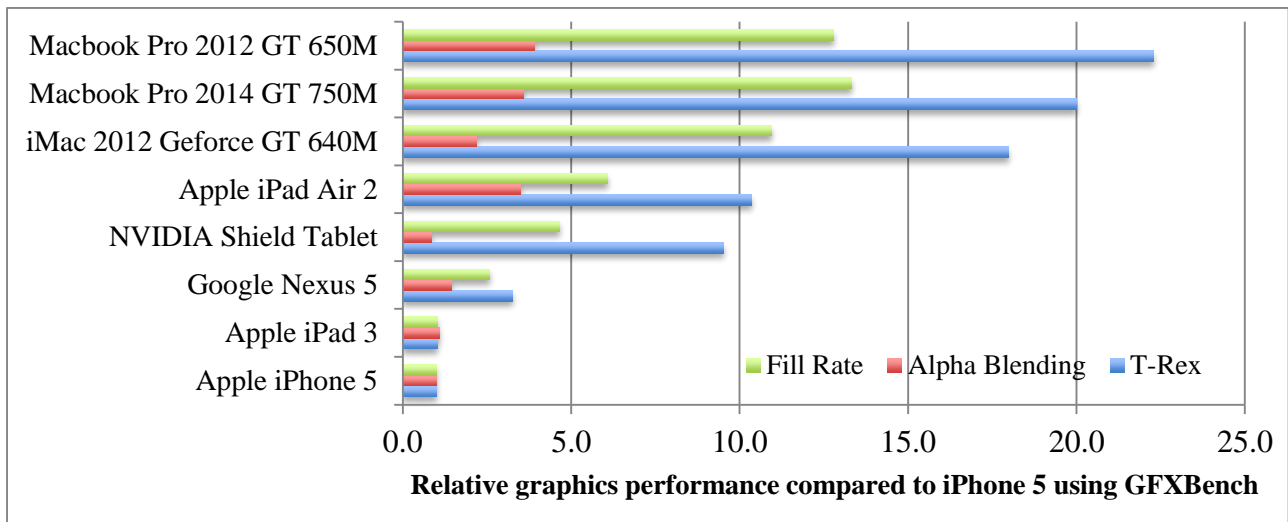


Figure 1 Comparing graphics performance between mobile devices and PCs

iPhone 5 is set as base-level of 1.0 and a result of 2.0 means a double running time performance compared to the iPhone 5. Results are gathered for three different GFXBench test cases: Fill Rate, Alpha Blending and T-Rex. Alpha blending test renders semi-transparent quads using high-resolution uncompressed textures. Fill rate measures texturing performance by rendering four layers of compressed textures. T-Rex measures realistic game like content. The fastest mobile device iPad Air 2 is only half slower than the Macbook Pro 2014 model.

Advances in mobile graphics performance and the graphics APIs allow new and complex graphics algorithms to be implemented in real-time. A computationally highly demanding field of computer graphics is volume rendering. It is used to simulate realistic fluids and clouds in computer games. Volume rendering is mostly used in the field of medical visualization. Enabling volume rendering on mobile device would allow a doctor to view MRI scans with a tablet device while visiting a patient.

In this thesis the basics of volume rendering algorithms will be covered. Existing research literature that applies volume rendering on mobile devices is reviewed. Finally, the thesis will evaluate performance of multiple volume rendering algorithms on the latest mobile GPUs.

## 2. Volume Rendering

Volume rendering is a set of techniques to project 3D volumetric data to a 2D display. Volumetric data consist of voxels which have the same size. Typical low-resolution volume dataset is 256 x 256 x 256 voxels. Voxels can be imagined as tiny cubes that represent a single value.

The volume dataset can be a result of sampling an object in three dimensions. One example of this is an organ scan using a MRI (magnetic resonance imaging) machine. Other medical equipment that produce volumetric data are CT (computer aided tomography) and PET (positron emission tomography) machines.

Voxel data can be acquired by transforming a 3D triangle mesh into voxels. Triangles are the basic primitive of computer graphics and widely used. Typically surfaces are presented with a set of triangles. This process of converting a triangle mesh to voxels is called voxelization (2). Voxelization processes all the voxels of the volume dataset and defines if a voxel is inside or outside the triangle mesh.

In order to draw a volume dataset into the display indirect or direct volume rendering methods can be used.

### 2.1. Indirect volume rendering

Indirect volume rendering methods transform the volumetric data into a different domain. Typically data is transformed into polygons that represent a single surface. High numbers of polygons could be generated for complex volume datasets. A triangle is a basic primitive for the GPUs and therefore polygons need to be converted into triangles if more than three vertices are used per polygon. Volume rendering is done by rendering the generated triangle list from the volumetric transform. Indirect volume rendering methods work well for the older generation GPUs as texture rendering support is not required. However, rendering a high number of triangles on older generation GPUs might not be feasible as the GPUs cannot handle millions of triangles in real-time. Problems with the indirect volume rendering approach are discussed in the following chapters.

#### 2.1.1. Contour tracing

Contour tracing is an indirect volume rendering method first introduced by Keppel (3) and later refined by Fuchs et al. (4). Contour tracing tracks contours of each volume slice and then connects the contours with triangles in order to form a surface representation of the volumetric data. Contours and the connected triangles are illustrated in Figure 2.

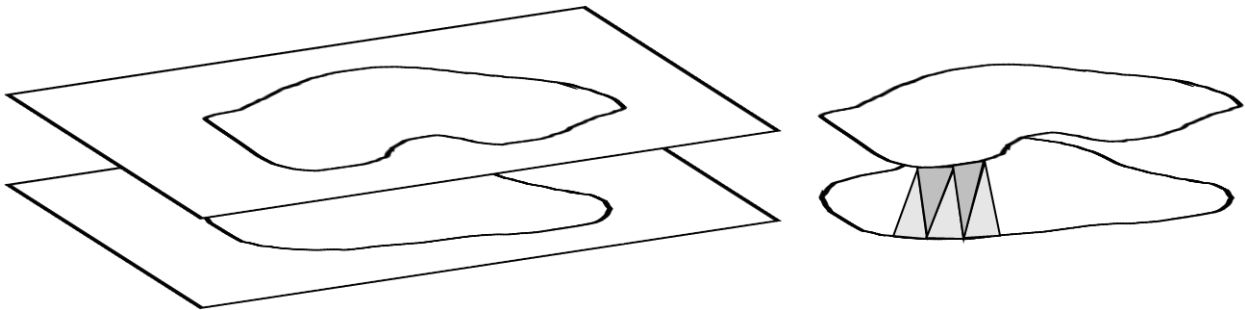


Figure 2 Volume slices with contours (left) and contours connected with triangles (right)

Contour tracing includes four steps: segmentation, labeling, tracing and rendering. In the segmentation step closed contours in 2D slices are detected with image processing methods and a polyline presentation is generated for the contour. In the labeling step, different structures are identified and labeled. Different labels could be created from different organs in human body for example. In the tracing step labels are used to connect the contours that represent the same object in adjacent slices. Triangles are produced as an output of the tracing step. Finally, in the last step triangles are rendered to the display.

There are several disadvantages and problems with contour tracing. If the adjacent slices have different amount of contours it can be difficult to determine which contours should be connected together. Similarly, contours that need to be connected could have different amounts of polyline points. The problem with this case is that it is hard to determine which polyline points should be connected together in order to form the triangle list.

Each contour could have a transparency value and therefore transparency could be supported by the contour tracing algorithm. However, all the triangles would need to be rendered in a back to front order and this would require sorting a large number of triangles in real-time. The sorted triangles need to be split into pieces in the cases where triangles overlap with each other.

### 2.1.2. Cuberille

Cuberille algorithm was first introduced by Herman et al. (5) and it uses a simple method of transforming the volumetric data into tiny opaque cubes. Each voxel from the volumetric dataset can be presented with a cube geometry. Each cube can be presented with 8 vertices that form 12 triangles. Each cube is the same size as the voxels in the volumetric dataset are equally spaced.

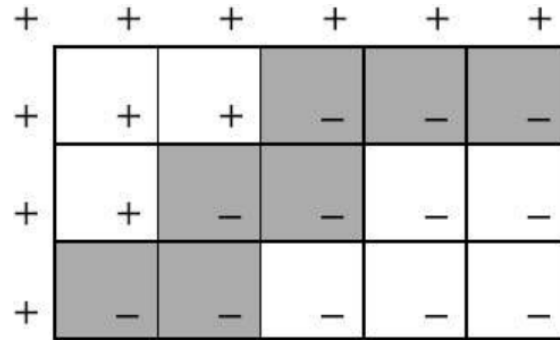


Figure 3 Cuberille algorithm produced opaque cubes marked with gray color.

In the cuberille algorithm binary segmentation is created for the volume dataset. Binary segmentation defines if a voxel is inside or outside the surface. In Figure 3 cube voxels that are inside the surface are marked with a negative sign and a positive sign is given to the voxels that are outside the surface. The process continues by searching the voxels that are in the boundary of the surface. Boundary voxels are marked with gray color in Figure 3 and opaque cubes are generated into these positions.

The benefit of the cuberille approach is that lightning can be easily added as surface normals are determined for the cube faces. Cuberille algorithm works well with GPUs as triangle primitives are used and no texturing hardware is needed because the generated cubes are colored only with one color. Cubes can easily be rendered in the correct order using the z-buffer technique. The z-buffer is an additional buffer to the framebuffer that stores a distance to the viewer. The rendering pipeline then determines if a pixel is or is not drawn using a depth test operation, which compares the value fetched from the z-buffer with the distance to the viewer. If the distance to the viewer is smaller than the value in the z-buffer, the pixel is drawn and the distance is stored to the z-buffer.

Images rendered with the cuberille can look very blocky. However, visual quality can be improved by subdividing cubes into smaller cubes. Subdivision can continue as far as 1-pixel sized cubes and therefore better visual quality is achieved. However, the performance is significantly decreased due to a high number of additional cubes.

### 2.1.3. Marching cubes

The marching cubes algorithm (6) improves the cuberille algorithm by directly generating triangles from the volume dataset representation. Like the cuberille algorithm, marching cubes uses a binary segmentation for the volume dataset. Marching cubes algorithm takes eight neighbor voxels and based on the voxel values creates triangles. Only a binary value inside or outside the surface is determined for each of the eight voxels. These eight binary values represent cube corners. Eight binary values can also be presented with one 8-bit index value. Based on this value, a different set



of triangles is produced as an output. A total of 256 different triangle lists can be produced out of the 8-bit value. However, based on the symmetry and other properties, 15 unique triangle sets can be identified. These unique triangle sets are illustrated in Figure 4.

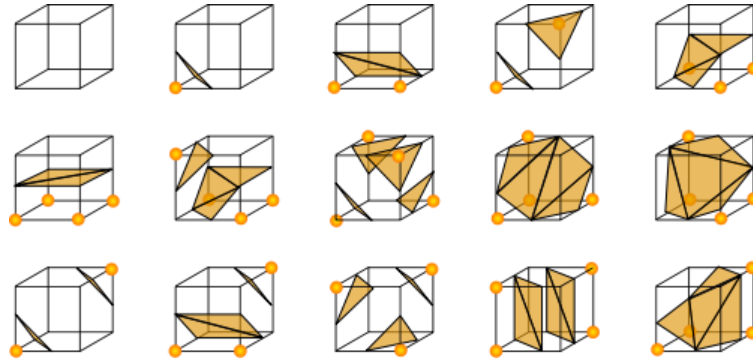


Figure 4 Unique cases for the marching cubes algorithm where a dot in the figure is a vertex inside a surface

The implementation of the marching cubes algorithm is relatively simple. The implementation only needs a 256 element lookup table that covers all the triangle cases. Index is generated by the marching cubes algorithm which is used to fetch the correct set of triangles from the lookup table.

In marching cubes algorithm there are some corner cases where invalid output is generated. This output can be seen in the rendered images as holes in the surface. This loss of accuracy is visible if the volume dataset has small and noisy details.

Memory consumption can be high for complex datasets when numerous triangles are generated. Assuming each vertex uses 32-bit floating point coordinates, one triangle would consume 36 bytes. Depending on the volume dataset if 1% of a  $256 \times 256 \times 256$  volume dataset generated an average of 2.5 triangles, the total number of triangles generated by the marching cubes algorithm is over 400000 triangles. Storing this amount of triangles takes over 14 MBs. This amount of memory consumption is high, as the original volume dataset is 16 MBs. The pre-allocation of a triangle list buffer is difficult, as the total number of triangles generated by the marching cubes algorithm is unknown. However, using multiple smaller buffers eliminates the problem.

#### 2.1.4. Marching tetrahedra

In order to avoid some of the corner cases where marching cubes produce errors, marching tetrahedra algorithm (7) was developed. In the marching tetrahedra algorithm, the cube is decomposed into five tetrahedra. The cube decomposition is illustrated in Figure 5.

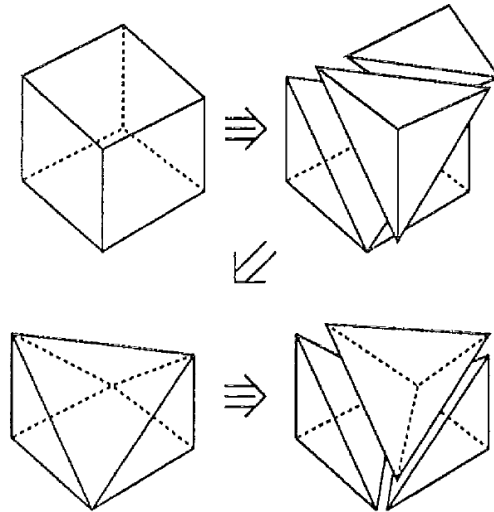


Figure 5 Decomposition of a cube into five tetrahedra

Each composed tetrahedron includes four vertices. For each vertex, testing is done if the volume data at the vertex position is inside or outside the volume isosurface. Four vertices produce up to 16 different configurations. Triangles are then created inside the tetrahedron similarly to the marching cubes algorithm. Large lookup tables are not needed for the marching tetrahedra algorithm, since only three unique cases can be identified from the 16 possible configurations.

The marching tetrahedra algorithm produces better visual quality compared to the marching cubes algorithm, because more triangles are produced. However, more memory is needed for storing the triangles.

## 2.2. Direct volume rendering

Direct volume rendering algorithms do not convert the volumetric data into a surface model but directly perform the rendering from the volume dataset. The benefit of the direct volume rendering approach is that no additional memory storage is needed for temporary buffers. Compared to the indirect volume rendering, direct volume rendering does not need a heavy preprocess transformation step. Several direct volume rendering algorithms exist and the most popular algorithms used are texture slicing and ray casting.

### 2.2.1. Shear-warp factorization

Shear-warp factorization volume rendering algorithm is proposed by Lacroute et al. (8). Shear-warp algorithm is optimized for CPU accelerated volume renderers. The algorithm is similar to 2D texture slicing and uses slices of textures and is an object-order method. The algorithm is illustrated in Figure 6.

The algorithm consists of three separate steps: shear, project and warp. In the shear step volume data is transformed to the sheared object space by translating and resampling each slice. In the project step, slices are composed together in front to back order. This step generates an intermediate image in the sheared object space. The result of the project step is warped and therefore the last warping step is needed to transform the warped image to the image space in order to get the correct final image.

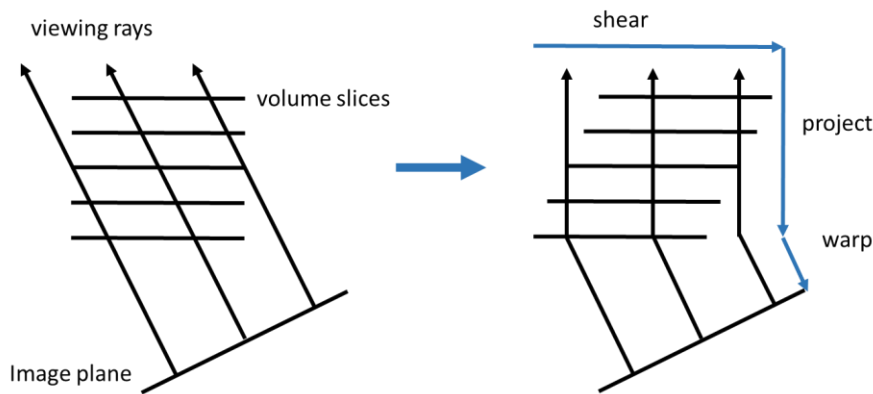


Figure 6 Shear warp algorithm illustrated

Benefit of the Shear-warp algorithm is that it suites well for CPU based renderers. CPU optimizations can be done as the sheared rays are parallel. Similarly, memory access patterns are better when the rays are parallel. Disadvantage of the shear-warp algorithm is decreased image quality due to the intermediate image used.

### 2.2.2. Splatting

Splatting volume rendering algorithm is proposed by Westover (9). Typically, in forward mapping methods like the ray casting algorithm, estimation is made which part of the input volume dataset affects a given pixel. Splatting is a backward method where estimation is made how many display pixels each voxel contributes. This is done by “splatting” each voxel to the image plane that is displayed on the screen. The process is similar to throwing a snow ball to a window.

In the first step of splatting it is determined in which order the volume should be traversed. This can be done by calculating the transformed cube corner coordinates and determining which corner coordinate is closest to the viewer. Voxels are then splatted in the order where the closest voxels to the viewer are splatted first one voxel slice at a time.

In the next step voxels are projected to the image plane and it is calculated how much each voxel contributes to the image plane. A round contribution filter with a circular kernel like the Gaussian one used in a case of parallel projection rendering. However, for the perspective projection, the

ellipse forms of varying shape must be used. Finally, front to back blending is used as a composition method.

### 2.2.3. Texture slicing

Texture slicing algorithms directly utilize the GPU's texturing hardware. Slicing algorithms with 2D texture fetching were introduced by Cabral et al. (10) and 3D textures were utilized first by Wilson et al. (11).

2D texture slicing algorithm renders each volume slice to the screen as a quad using two triangles. In order to get the correct transparency, rendering is done in back to front order. In back to front order rendering all the rendered quads need to be depth sorted and the drawing happens first by rendering the quad that has the largest distance from the viewer. Rendering continues a quad at a time and lastly the nearest quad to the viewer is rendered. The stack of volume slices is illustrated in Figure 7.

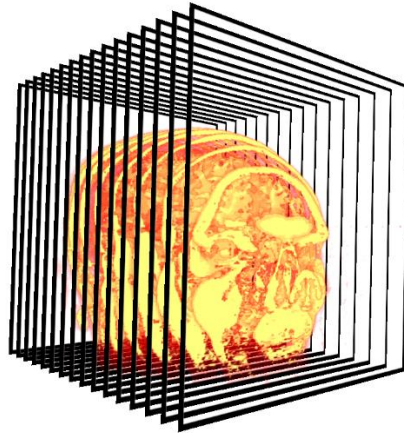


Figure 7 Texture slicing illustrated with 16 slices

In 2D texture slicing each volume slice is stored in memory as a 2D texture. Limitation of the 2D texture slicing is that one rendered quad matches one slice in the volume dataset. Therefore, sampling rate can only be increased by introducing new texture slices to the volume dataset. In practice, doubling the sampling rate also doubles the amount of memory storage needed for the volume slice textures.

Holes become visible in the rendered images using the texture slicing algorithm when the volume slices are rotated more than 45 degrees. However, this problem can be solved by allocating three sets of volume slice stacks each of which represents x, y and z coordinate axis. Unfortunately, three times more memory is required to store the new volume stacks. Rendered holes can be avoided by selecting the correct volume stack based on object rotation. Benefit of the 2D texture slicing is that it works on all the GPUs that support basic 2D texturing.

2D texture slicing can be improved for the GPUs that support multi-texturing. In multi-texturing the GPU hardware is able to fetch two or more textures and perform blending operations between them. In volume rendering, multi-texturing can be used to fetch between two slices of volume data. Therefore, the sampling rate can be increased without any extra memory penalty. However, there is a performance penalty in multi-texturing as two separate textures need to be fetched and blended together.

Extra 3x memory consumption with the 2D texture slicing can be solved by using 3D textures. With 3D textures GPU hardware directly fetches the voxels from a 3D texture. The operation is similar to the multi-texturing where two texture layers are fetched and blended. However, the benefit of the 3D texturing is that actual 3D texture coordinates can be used to fetch the voxels. This allows view aligned volume slices were the 3D texture coordinates are rotated and not the object itself as in the 2D texture slicing algorithms. Increased quality is achieved by using the view aligned slices.

#### 2.2.4. Ray casting

Ray casting, one of the most used volume rendering algorithm was first introduced by Levoy (12). The algorithm uses a ray from a viewer to the volume. Along the ray, voxels are sampled at constant steps and composed together to form the final pixel color to the display. Typically, voxel sampling and composition is done in front to back order. Figure 8 illustrates the function of the ray casting algorithm. For each display pixel a ray from the viewer (V) to the volume dataset is calculated. The ray enters the volume at location P1 and exits the volume at P3 location. There is no volume data before the point P1 and after the point P3. Therefore, ray casting stepping can start from the point P1 and stop at the point P3. Semi-transparent area in Figure 8 is marked with blue color and fully opaque area with yellow color. Ray enters the semi-transparent area at the point P2. Before this volume is fully transparent in the illustration. However, no visibility optimizations are made in the basic version of the ray casting algorithm and all the voxels are traversed.

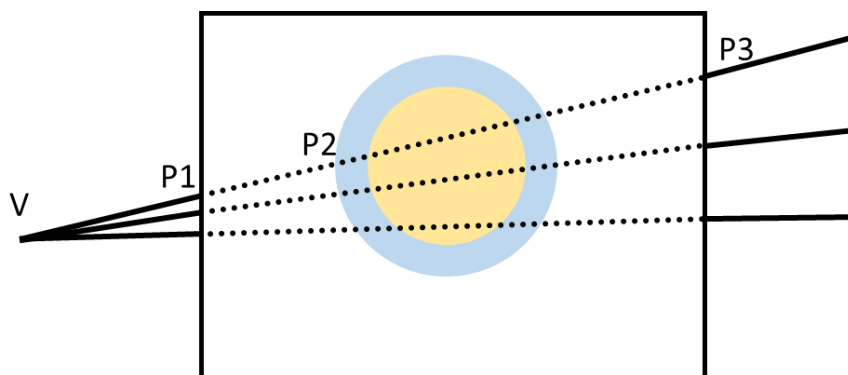


Figure 8 Ray casting steps illustrated

Points where the ray enters the volume dataset and exits can be calculated with the box and ray intersection mathematics. However, a simple method for programmable GPUs was proposed by Kruger et al. (13). This acceleration method uses color coded coordinates. Color coding is illustrated in Figure 9.

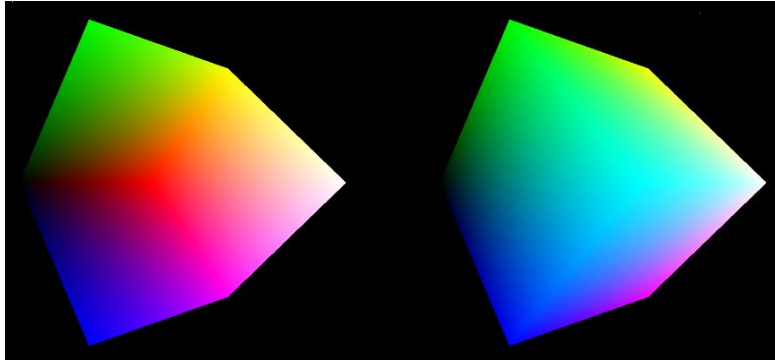


Figure 9 Color coded ray casting start (right image) and end (left image) coordinates

In the acceleration method back faces of the cube (left image) are first rendered to an off-screen texture. Then front faces of the cubes are normally rendered to the display and the programmable shader code is used to fetch the color value from the off-screen texture. This color value from the off-screen texture defines where the ray casting ends and the front color defines where the ray casting starts in the volume dataset. A ray direction can be simply calculated by subtracting the end from the start location. The shader program then loops through the volume dataset with constant stepping. During the looping voxel values are composited into a color value that is displayed in the screen once the loops is finished.

The ray casting implementation requires 3D texture support in order to access the volume dataset with 3D texture coordinates. 3D textures can be emulated with a 2D mosaic texture in the case where 3D textures are not supported by the GPU hardware. However, performance is lost in the emulation.

### 2.3. Classification

The classification step (12) in the volume rendering allows to see inside of an object and explore its structure instead of only visualizing the surface of the object. The step involves mapping voxel scalar values to opacity and color values.

Classification is performed by means of a transfer function. Input for the transfer function is the voxel value and output is the color value. Figure 10 illustrates how two different transfer functions produce very different end results. In the example, engine volume dataset voxel values represent different material densities. In the left image low density values are mapped to a white opaque color

and high density values are made fully transparent. Opposite to this in the right image high density areas are made visible with blue color and the low density value are given a transparent value.

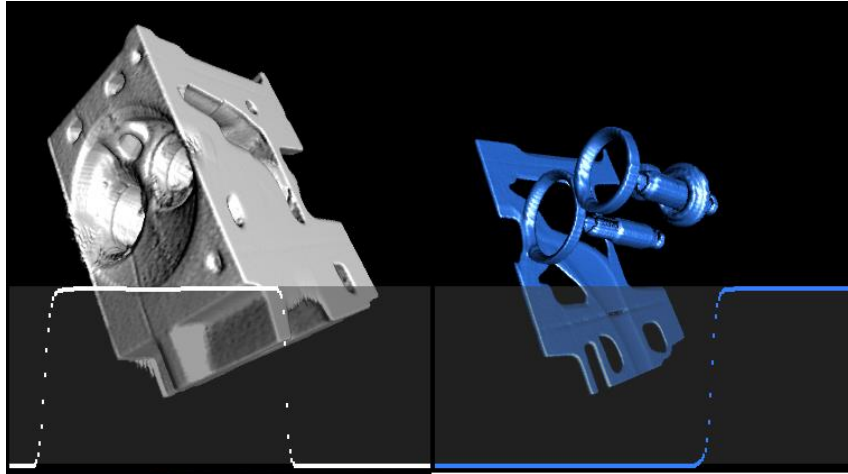


Figure 10 Two different transfer functions visualized for the same volume dataset

There are two options when the transfer function can be applied. In the pre-classification case transfer function is applied to the full 3D volume texture as a preprocessing step. The other option is the post-classification case where the transfer function is applied to each voxel when they are accessed. This is typically done with a lookup table in the shader code.

## 2.4. Composition

Color composition is needed for volume rendering in order to mix voxel color values together. Several different color composition models are introduced by Porter (14). Most useful color composition models defined for the volume rendering are over and under operations.

Over operation is mostly used with the slicing algorithm and back to front ordered rendering. It composites a color over another color with a specified alpha value. The equation for the over operation is given by (2.1) where two colors  $C$  and  $C_{in}$  blended with alpha value produce the  $C_{out}$  color value. Calculation is performed for each of the red, green and blue color channels.

$$C_{out} = (1 - \alpha) C_{in} + C \quad (2.1)$$

Under operation is typically used with ray casting algorithm and also with other rendering methods that operate in the front to back order. Equation for the under operation is given in (2.2). Difference to the over operation is that alpha values need to be calculated and stored in the under operation.

$$\begin{aligned} C_{out} &= C_{in} + (1 - \alpha_{in}) C \\ \alpha_{out} &= \alpha_{in} + (1 - \alpha_{in}) \alpha \end{aligned} \quad (2.2)$$

Other useful composition methods exist for the volume rendering. The first operation as the name suggest takes the first non-transparent color value and uses it as an output for the pixel. Average operation takes average of all the voxel values that contribute to the pixel value. Maximum intensity projection (MIP) composition method takes the maximum value of all the values that contribute to the pixel value. MIP method is mostly in medical volume visualizations where contrast agent is injected into blood vessels.



### 3. Local illumination models

Local illumination models add more realism to the volume rendered images by calculating shading for the rendered images. The term local refers to the fact that in the local illumination lighting is only calculated locally and other objects or even the object itself does not affect the end results. Therefore, shadows are not produced by the local illumination. However, as the local illumination does not take the surroundings into consideration it is a fairly simple and fast method to add more realism to the rendering.

#### 3.1. Gradient

In order to calculate illumination, a surface normal is needed. Typically it is easy to calculate normals for a triangle mesh but a different approach is needed for a volume dataset. Normals can be calculated for a volume dataset using gradients (15). A gradient is a first order derivative of the volume scalar field. Typically, with real-time volume rendering algorithms either forward or central gradient estimation is used because of their simplicity. However, a high quality gradient estimation can be used when the normals are pre-calculated to a 3D texture.

In the forward gradient estimation the current voxel value and the next voxel value are used to calculate the gradient. The equation for the forward gradient estimation is given in (3.1). The forward gradient estimation uses four memory fetches per voxel element.

$$\nabla f(x, y, z) \approx \begin{pmatrix} f(x + h, y, z) - f(x, y, z) \\ f(x, y + h, z) - f(x, y, z) \\ f(x, y, z + h) - f(x, y, z) \end{pmatrix} \quad (3.1)$$

A better result can be achieved by using the central difference of voxel values. In the central gradient estimation the previous and the next voxel are used to calculate the gradient. The central difference can be calculated with six memory fetches and therefore is computationally bit more expensive compared to the forward difference method.

$$\nabla f(x, y, z) \approx \begin{pmatrix} f(x + h, y, z) - f(x - h, y, z) \\ f(x, y + h, z) - f(x, y - h, z) \\ f(x, y, z + h) - f(x, y, z - h) \end{pmatrix} \quad (3.2)$$

Using additional memory storage and storing normals to a 3D texture allows high quality normals to be calculated as more time in the calculation is feasible. Sobel (16) introduced gradient estimation in the 2D image domain that can be applied to 3D and volume rendering.

Convolution in 3D can be used to estimate the gradient. Convolution matrices are presented in Equations (3.3), (3.4) and (3.5). Different matrices are used for each X, Y and Z axis.

$$X1 = \begin{pmatrix} -1 & 0 & 1 \\ -3 & 0 & 3 \\ -1 & 0 & 1 \end{pmatrix}, X2 = \begin{pmatrix} -3 & 0 & 3 \\ -6 & 0 & 6 \\ -3 & 0 & 3 \end{pmatrix}, X3 = \begin{pmatrix} -1 & 0 & 1 \\ -3 & 0 & 3 \\ -1 & 0 & 1 \end{pmatrix} \quad (3.3)$$

$$Y1 = \begin{pmatrix} -1 & -3 & -1 \\ 0 & 0 & 0 \\ 1 & 3 & 1 \end{pmatrix}, Y2 = \begin{pmatrix} -3 & -6 & -3 \\ 0 & 0 & 0 \\ 3 & 6 & 3 \end{pmatrix}, Y3 = \begin{pmatrix} -1 & -3 & -1 \\ 0 & 0 & 0 \\ 1 & 3 & 1 \end{pmatrix} \quad (3.4)$$

$$Z1 = \begin{pmatrix} -1 & -3 & -1 \\ -3 & -6 & -3 \\ -1 & -3 & -1 \end{pmatrix}, Z2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, Z3 = \begin{pmatrix} 1 & 3 & 1 \\ 3 & 6 & 3 \\ 1 & 3 & 1 \end{pmatrix} \quad (3.5)$$

A high number of calculations are needed in order to perform the convolution. It takes 54 multiplications and 27 memory fetches for each calculated gradient. It would not be feasible to do this amount of calculations in real-time on mobile devices. However, it is possible to calculate the normals in the volume dataset loading phase or then store the pre-calculated results to a file.

Different quality is provided with forward, central and pre-calculated Sobel normals. Quality difference is illustrated in Figure 11. Even though, there is little difference between the options, the pre-calculated Sobel normals provide the best quality.

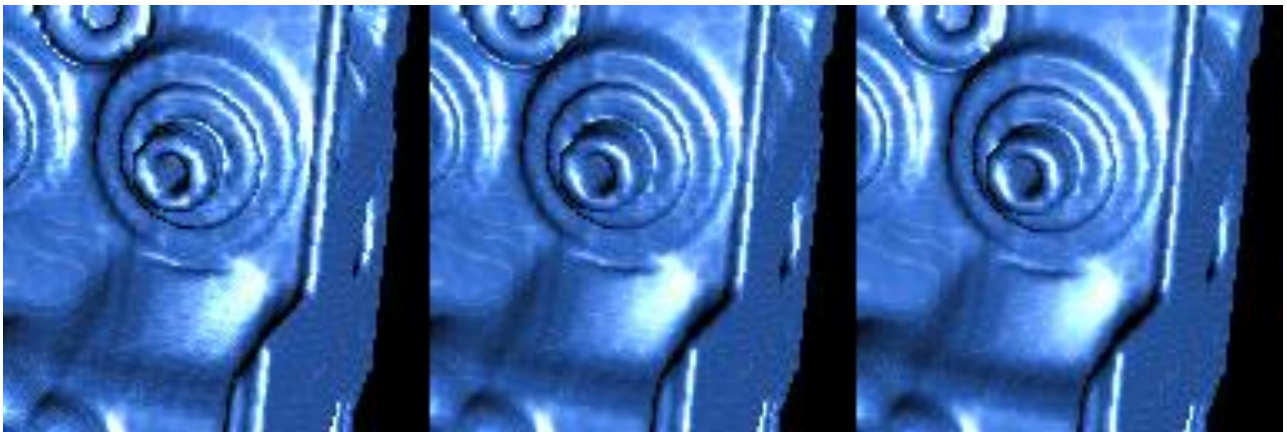


Figure 11 Gradient estimation quality comparison between forward difference (left image), central difference (middle image) and pre-calculated Sobel normals (right image).

### 3.2. Phong

A popular illumination model in computer graphics is the Phong illumination model (17). It is also suitable for volume rendering to calculate the local illumination. Calculation is done with Equations (3.6), (3.7), (3.8) and (3.9). Needed vectors are illustrated in Figure 12.

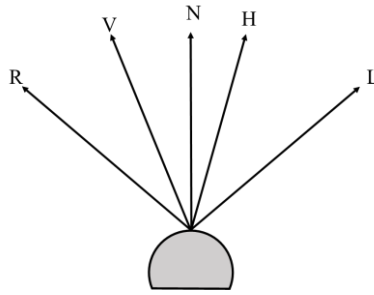


Figure 12 Surface and the needed vectors for the illumination calculations

As previously mentioned normal vector  $N$  can be calculated using gradients in the volume rendering. Vector  $L$  is a direction from the voxel to the light position and vector  $V$  is to the viewer. Reflection vector  $R$  can be calculated using the normal vector  $N$  and the light vector  $V$ .

$$R = 2 * (N \cdot L)N - L \quad (3.6)$$

$$I_d = N \cdot L \quad (3.7)$$

$$I_s = (R \cdot V)^\alpha \quad (3.8)$$

$$I = I_a + I_d + I_s \quad (3.9)$$

Illumination  $I$  can be calculated from the ambient  $I_a$ , diffuse  $I_d$  and specular  $I_s$  terms. Ambient defines a constant ambient light color and the light position or the viewer does not affect the result. The diffuse term uses the normal  $N$  and the light vector  $L$  and is calculated using a dot product operation. Dot product is the angle between these two vectors and the value of the dot product is 1.0 when the angle between the vectors is zero. Therefore, the brightest value can be achieved for the diffuse term when the normal is pointing to the light direction.

Specular term is composed from the angle between the reflection and the view vector. Shininess of the surface can be controller with the exponent term. High exponent values produce smaller spots of specular light. The diffuse and the specular term are visualized in Figure 13. Even the plain diffuse term gives more depth perception to the rendered image. Specular term adds white spot lights to the final rendering.



Figure 13 Illumination effect to the rendering (left image: no illumination, middle image: illumination with the diffuse term, right image: illumination with the diffuse and the specular term)

### 3.3. Blinn-Phong

Blinn (18) created a variant of the Phong illumination model. In this illumination model half-vector  $H$  is calculated using the light  $L$  and view  $V$  vectors. The specular term is then modified and the angle between the half-vector and the normal is used to calculate the specular term. Otherwise, the Blinn-Phong illumination is calculated the same way as Phong illumination. The Blinn illumination half-vector Equation is presented in (3.10) and the specular term in (3.11).

$$H = \frac{L + V}{|L + V|} \quad (3.10)$$

$$I_s = (N \cdot H)^\alpha \quad (3.11)$$

However, Phong and Blinn models do not produce the same visual appearance. The reason for this is that the angle between  $N$  and  $H$  is not the same as  $R$  and  $V$ . This can be seen from Figure 12 where the angle between  $N$  and  $H$  is smaller than the angle between  $R$  and  $V$ . Therefore, the specular exponent can be adjusted in order to achieve similar results with both the Blinn and the Phong illumination models.

## 4. Global illumination models

Global illumination models increase the visual perception as shadows, reflections and refractions are visualized. In the volume rendering global illumination model, the surrounding voxels and the light sources affect each voxel. The complexity of the global illumination calculations is extremely high and most of the algorithms cannot be run in real-time. Popular shadowing techniques for volume rendering are shadow rays, shadow mapping and half-angle slicing.

### 4.1. Shadow rays

Adding shadows to volume rendering can trivially be achieved with shadow rays (15). By casting a ray from each voxel step to the light source, it is determined whether the current voxel sample is in a shadow or not. The benefit of the shadow ray technique is that transparency is supported. Transparency can be determined by calculating the opaque factor along the shadow ray. A voxel is completely in a shadow, if fully opaque voxels are found during the shadow ray steps and partially in the shadow, if semi-transparent voxels are sampled by the shadow ray.

The disadvantage of the shadow ray technique is the increased complexity. The algorithm has the complexity of  $O(nm) = O(n^2)$ , where  $n$  is the total number of sample steps in the ray casted from the viewer and  $m$  is the number of the shadow ray steps.

The complexity can be reduced to  $O(n)$  if the shadow rays are only casted once for each voxel and the results are stored to a shadow volume texture. However, changing the light source position compared to the volume dataset triggers regeneration calculation of the shadow volume texture.

### 4.2. Shadow mapping

Shadow mapping was presented by Williams (19). It is a simple technique where a shadow map is rendered to a texture from the light source view. The shadow map is placed between the object and the light source. Shadow rendering stores distance values to the light source for each shadow map pixel.

Shadowing for each voxel is done by comparing the voxel distance to the light source and to the stored distance in the shadow map. If the distance value for the voxel is higher than the distance value stored in the shadow map, the voxel is in the shadow.

The complexity of the shadow mapping algorithm is  $O(n)$  as only one extra rendering pass for the volume dataset is needed to render the shadow mapping texture.

However, the disadvantage of shadow mapping is that it works only with the opaque volume datasets as only one distance is stored to a shadow map.

### 4.3. Half-angle slicing

Kniss et al. (20) propose an algorithm called half-angle slicing, which uses an additional pixel buffer to store accumulated light attenuation. The algorithm defines a slicing axis, which is halfway between the light and the view direction. The slicing axis allows simultaneous rendering from the light and the viewer direction. The slicing axis is illustrated in Figure 14.

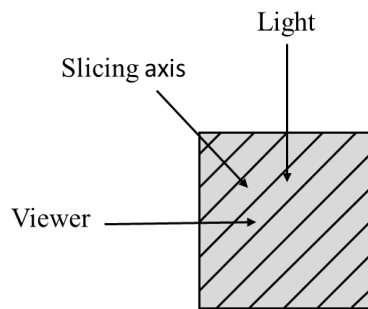


Figure 14 Half-angle slice axis for light transport.

Rendering is done iteratively in the half-angle slicing algorithm. First, a slice is rendered from the viewer's point of view and voxel brightness is modulated by the results from the light buffer. The same slice is then rendered from the light's point of view, to calculate the intensity of the light at the next slice.

The benefit of the half-angle slicing algorithm is that only one additional 2D texture is required to store to the light accumulation buffer and the algorithm works in real-time with  $O(n)$  complexity.

## 5. Optimization techniques

Optimization techniques are important in volume rendering as significant performance gain is due to be achieved. Typically, optimization techniques reduce the amount of voxels that need to be evaluated. This is done by skipping fully transparent areas of the volume dataset or exiting the ray casting in the case of an opaque voxel. These techniques provide faster rendering times. However, other optimization techniques like the bricking optimizes the memory consumption of the volume dataset.

### 5.1. Empty space skipping

It is beneficial if fully transparent voxels can be skipped in the ray casting algorithm in order to allow fewer calculations and therefore faster rendering times. The algorithm for skipping the empty voxels for the ray casting algorithm is presented by Levoy (21).

Optimization is done by introducing a pyramid of binary volumes. For each pyramid level, one bit value determines whether a cell in the pyramid is empty or nonempty. In the case where a cell is empty, the whole cell can be skipped. Skipping is done by calculating the intersection between the casted ray and the cell. If a cell is not empty, testing is done for a lower level in the pyramid. The algorithm is illustrated in Figure 15.

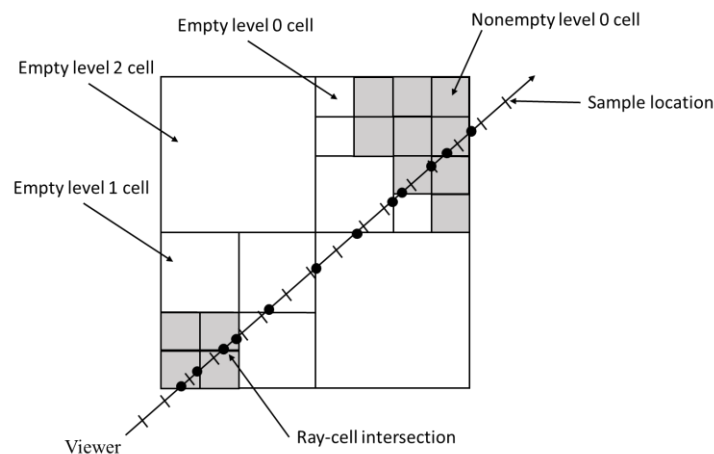


Figure 15 Acceleration of empty area skipping for ray casting

The disadvantage of the algorithm is that more processing is needed for fetching the values from the pyramid hierarchy and it depends on the volume dataset how well the algorithm works. Measurements done by Levoy indicated that up to 2-5x speed increased is possible with the algorithm. However, the implementation was made for CPUs and it is unknown how well the algorithm would work on a GPU.

Another ray casting speedup algorithm is introduced by Zuiderveld et al. (22). It uses a 3D distance transform to skip the empty areas of the volume dataset. The distance transform is a preprocessing step before the actual rendering. In the distance transform step, distance to semi-transparent or opaque voxels is calculated from each empty voxel. The distance information is then stored to a new distance volume texture. Ray casting can then be accelerated by using the distance values from the distance volume. Acceleration is illustrated in Figure 16.

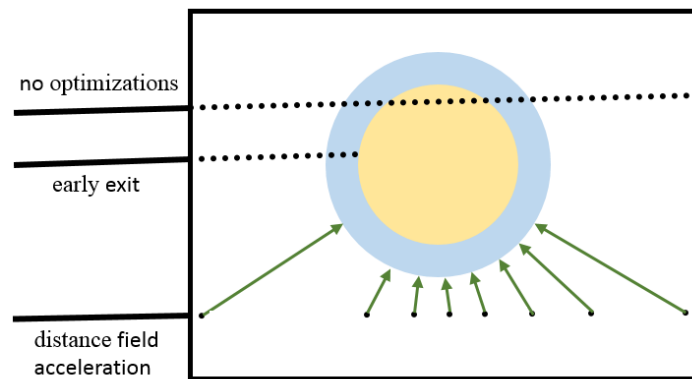


Figure 16 Ray casting optimization techniques

Normal implementation (no optimizations case in Figure 16) uses 43 ray casting steps inside the volume dataset. However, using a distance volume only takes 8 steps as the ray totally bypasses the opaque voxels of the volume dataset. For each step in the ray casting, a distance value is fetched from the distance volume. The value indicates how many steps in the ray casting can be skipped safely without intersecting to the opaque volume dataset. In Figure 16 large steps can be skipped, when the ray enters the volume as the distance to the opaque voxels is high. Once the stepping is closer to the opaque parts of the volume, smaller steps can be skipped.

The benefit of the 3D distance transform is that skipping is done efficiently and a lot of unnecessary calculations can be avoided. However, the created distance volume takes additional memory storage and the actual pre-calculation step that calculates the 3D distance transform can be time-consuming.

## 5.2. Early ray termination

Another technique for performance optimization can be realized in the ray casting algorithm when the ray casting is done in a front to back order. In front to back ray casting new opacity value is calculated in each ray casting step. Once the opacity value gets close to a 1.0 value, ray casting can be stopped as further voxel sample points along the ray do not contribute to the final pixel value.

An early ray termination is illustrated in Figure 16 where a ray is terminated when the ray first reaches the opaque area of the volume dataset.



### 5.3. Adaptive refinement

In some cases volume rendering speed can be decreased especially with large volumes or volumes that are rendered using complex illumination models. In these cases, an adaptive refinement process presented by Levoy (23) can be used.

The adaptive refinement for volume rendering is done in multiple passes. The image is first rendered only by casting a few rays and the resulted low resolution image is displayed using filtering. The refinement process continues by casting more rays and each time a new image is displayed. The benefit of adaptive refinement is that the first few images are quickly shown to the user. The user can then decide to wait for the final image or adjust some parameters of the volume rendering and start a new rendering.

### 5.4. Bricking

Most of the GPUs have limited maximum size for the textures. Similarly, a total amount of texture memory is limited. In order to render large volume datasets a technique called bricking was invented (15).

Bricking splits the volume dataset into smaller bricks that can fit inside the texture memory. The benefit of bricking is that the storage of empty bricks is not needed. Typically, the resolution of the brick can be as small as  $16 \times 16 \times 16$ . On systems that have a limited amount of texture memory, but high amount of CPU memory, it is possible to use the CPU memory as a cache for storing the bricks. In this case, bricks are streamed into the GPU memory whenever needed by the volume renderer. Even larger volume datasets can be rendered when streaming the bricks from the hard drive. However, the performance is decreased dramatically as streaming data from the hard drive is a slow operation.

A problem exists when reading voxel values from the brick textures. GPU hardware filtering is done so that neighbor voxel values are needed in order to calculate the weighted average between the voxels. Therefore, duplicate voxel values are needed on the brick boundaries.

An added benefit of bricking is that empty space skipping can easily be supported. Using empty space skipping with a large volume dataset and bricking is presented by Tong et al. (24).

## 6. Existing solutions

The first open standard OpenGL ES 1.0 (25) graphics programming API for mobile devices was introduced in 2003. First OpenGL ES 1.0 implementations did not support any hardware acceleration at all. Implementations were optimized as software based solutions. Later accelerated 3D graphics was introduced in Windows Mobile (2005), Symbian (2006) and iOS (2007) operating systems. However, wide adaptation of the API was only available in the first iPhone.

### 6.1. Interactive Volume Rendering on Mobile Devices

Early research on accelerating volume rendering on mobile devices was made by Moser and Weiskopt (26). They implemented volume rendering for Dell Axim X50v/X51v device. The device was running on Windows Mobile OS, had 64 MBs of memory and included a 640 x 480 resolution display.

Texture slicing method was used as the OpenGL ES 1.0 API did not support the programmable graphics pipeline. As 3D textures were not available on the OpenGL ES 1.0, 2D textures had to be used for the slicing algorithm.

As the device supported only 64 MBs of memory there were some difficulties to run high resolution volume datasets when a 64 x 64 x 64 resolution brain dataset was used in the research. There were also difficulties in the transfer function implementation as the transfer function requires dependent texture lookup. OpenGL ES 1.0 API supports 8-bit palette textures, however, the research indicated that OpenGL ES driver always converted palette textures to full 32-bit RGBA images. For this reason 32-bit RGBA textures were directly used. The disadvantage in this solution is that whenever transfer function updates we need to update the full volume texture. Similarly, the original 8-bit volume dataset needs to be stored in the memory and this increases the memory consumption.

The brain dataset was benchmarked in the full device resolution (640 x 480) and also in the lower 128 x 128 resolution. Lower resolution rendering used an offscreen texture which was drawn to the display whenever rendering occurred. Results indicated that interactive frame rates were not achieved as rendering a single frame took 652 ms in the full resolution and 101 ms in the lower resolution.

### 6.2. Practical Volume Rendering in Mobile Devices

Later OpenGL ES 1.0 evolved to OpenGL ES 2.0 (27) and a fully programmable graphics pipeline was introduced. This meant that it was possible to program the vertex transform using vertex shaders and each pixel output color could be programmed with pixel shaders. Unfortunately, core

OpenGL ES 2.0 API does not support 3D textures as it is a choice for hardware vendors if they want to support 3D textures or not. There are only few hardware vendors that support 3D textures on OpenGL ES 2.0 generation hardware. Qualcomm has been supporting 3D textures for their hardware from the beginning and it is available since Adreno 200 GPUs.

GPU programmability allowed that a wider set of volume rendering algorithms were possible with mobile devices. One of the first published volume renderer using OpenGL ES 2.0 API was made by Rodriguez and Alcocer (28).

Ray casting and volume slicing were implemented and detailed benchmarking results were available for multiple mobile devices. The ray casting algorithm used 3D textures and therefore their implementation was only available for Qualcomm Adreno GPUs. However, they were able to compare different devices and GPU architectures using 2D texture slicing. Also, 3D texture slicing was implemented as Adreno 200 / 205 GPUs supported that extension. Other GPUs in the research were NVIDIA's Tegra 2 and Imagination's PowerVR SGX 540.

It was discovered that most of the devices had GPU memory limitations and the 256 x 256 x 256 volume dataset was the largest they were able to load on all the tested devices.

As GPUs were not very efficient they implemented rendering in a high resolution (800 x 480) and in low resolution (480 x 240). The low resolution was used always when a user interacted with the volume data. Similarly different amounts of steps (64, 128, 256 and 512) were possible for the slicing and ray casting renderer. Interactive frame rates up to 34 fps were reached for Adreno 205 device with 64 steps and low resolution using the 2D texture slicing algorithm. Adreno 205 GPU performed the best in all benchmark test cases and was 2.8x faster than SGX 540 and Tegra 2.

Key results of the benchmarking data indicated that the 2D texture slicing algorithm was 5.6x faster than ray casting algorithm. An obvious difference between the slicing and ray casting algorithms is that in slicing the fixed function GPU hardware is used to perform the blending operation and in ray casting the shader code is used for the blending.

Similarly 2D texture slicing was 2.3x faster than 3D texture slicing. This indicates that the actual 3D texture fetching is 2x slower than 2D texture fetching. However, in the results 512 step count was missing for the 2D texture slicing algorithm. This indicates that no multi-texture approach was used in the 2D texture slicer.

### **6.3. Volume Rendering Strategies on Mobile Devices**

Typically OpenGL ES 2.0 based ray casting implementation uses 3D textures. However, as 3D texturing support is very limited, Noguera et al (29) solves this limitation by using a texture mosaic that is a simple solution to store 3D slices into one 2D texture. As an example the  $64 \times 64 \times 64$  volume dataset could be stored as 64 slices ( $8 \times 8$  2D grid) inside a  $512 \times 512$  2D texture. Even such a small volume dataset uses quite a high resolution 2D texture and there is a limitation how a large volume dataset can be stored inside a 2D texture.

Typically, the volume dataset size is  $256 \times 256 \times 256$  and this can be stored as a  $4096 \times 4096$  2D texture. However, OpenGL ES 2.0 specification only mandates that  $64 \times 64$  textures must be supported. In reality almost all mobile devices support  $2048 \times 2048$  textures but there might a problem when using large mosaic textures as the minimum of  $4096 \times 4096$  2D textures would be needed. An obvious problem with the mosaic texture is accessing the volume data. In order to access the volume data in the pixel shader code a slice location has to be calculated and this can be a heavy operation. Unfortunately, no shader code for the mosaic fetching was provided neither were any performance penalty estimations. Similarly a problem arises with mosaic texture filtering. As bilinear filtering uses neighboring texel samples from the texture, there should be enough room between each slice inside the mosaic texture or visual artifacts occurs when sampling the mosaic texture. Texture coordinates should also be clamped to 0,1 range before accessing the mosaic or wrong samples would be accessed.

In addition to ray casting, texture slicing is implemented. The slicing implementation uses an interesting technique originally introduced by Gelder and Kim (30). This technique solves the quality problem with the slicing algorithm that is caused by rotating proxy geometry for the volume. Especially for near 45 degree rotations typical slicing implementations suffer from bad quality. In order to avoid rotating proxy geometry it is possible to keep the proxy geometry aligned to the viewing direction and rotate the texture coordinates of the proxy geometry. This is illustrated in Figure 17.

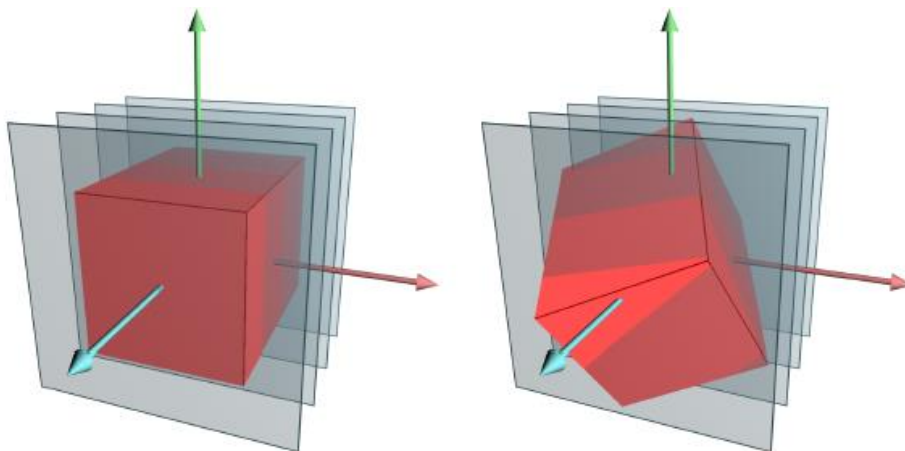


Figure 17 View direction aligned slices (left image) and rotated texture coordinates (right image)

A disadvantage of this solution is that proxy geometry around the volume dataset needs to be larger so the rotated volume dataset fits inside the proxy geometry. Texture coordinates are transformed inside the vertex shader. This should not cause any performance issues as the proxy geometry has only few vertices. However, the problem is the rotated texture coordinates in the pixel shader which could be out of the 0,1 range. Therefore any texel fetches outside the volume dataset needs to be ignored and this check must be done for each fetch from the mosaic. This is done simple by setting fetches from outside the volume to the voxel color that represents the fully transparent voxel.

Slicing and ray casting algorithms are benchmarked on iPad 2 and iPod 4<sup>th</sup> generation devices. Results include several different steps (10, 20, 40 and 80) in 1024 x 768 and 320 x 240 display resolutions. Results indicate that slicing is almost 2x faster than ray casting with 10 steps in the 320 x 240 resolution. However, when increasing the step count to 80 and benchmarking 1024 x 768 ray casting is slightly faster than slicing.

Quality is compared and results indicate that ray casting produced better quality over the slicing algorithm.

#### **6.4. Interactive 3D Image Processing System for iPad**

Volume rendering algorithms can be implemented without OpenGL ES API. A solution created by Yuan et al (31) uses only CPU to visualize the volume dataset. The implementation is run on iPad.

The maximum intensity projection (MIP) composition technique is implemented with memory optimization techniques. Memory is optimized by dividing the volume into smaller cells and creating optimal data structures that remove unused cells. Results indicate that total of 75% space was saved for the fruit fly volume dataset (512 x 512 x 108) and data reduction for the human brain dataset (181 x 217 x 181) was 43%.

Optimizing a volume data set with the cell removal algorithm takes 14213 ms for the fruit fly and 5682 ms for the human dataset. However, rendering is faster and 2022 ms (0.5 fps) is spent on rendering one frame for the fruit fly dataset and 1300 ms (0.8 fps) is spent on rendering human dataset. Clearly interactive rendering is not possible with the CPU implementation.

#### **6.5. Interactive visualization of volumetric data with WebGL in real-time**

OpenGL ES APIs are native C-language APIs. However, OpenGL ES APIs can be used via bindings to other programming languages. Nowadays, web browser runtimes operate on multiple

platforms and the clear benefit of running web apps is that no installation of the software is needed. Software can also be updated all the time on the server side.

Performance can be a problem with the web based applications as the applications are written using JavaScript that does not offer the same performance level as the native C/C++ APIs. However, it is possible to write 3D visualization applications using WebGL (32) API. WebGL API is based on the OpenGL ES 2.0 API and offers the same feature set. The latest web browsers support WebGL even on the mobile devices.

Volume rendering algorithms can be accelerated using WebGL API. Congote et al. (33) implements ray casting algorithm using WebGL API. Their work uses medical imaging and weather radar visualization datasets. The transfer function can be edited in real-time for both datasets.

Ray casting is implemented in two phases. First, back faces of a cube are rendered to an off-screen texture. The second pass renders front faces of a cube and fetches values from the rendered off-screen texture. Ray casting direction to the volume can be determined from the value difference between front and back faces.

WebGL has the same limitations as OpenGL ES 2.0 API and 3D textures are not supported. Therefore, the texture mosaic is used when fetching voxels in the ray casting algorithm. The texture mosaic is illustrated in Figure 18. The mosaic includes 97 slices in a 10 x 10 image grid.

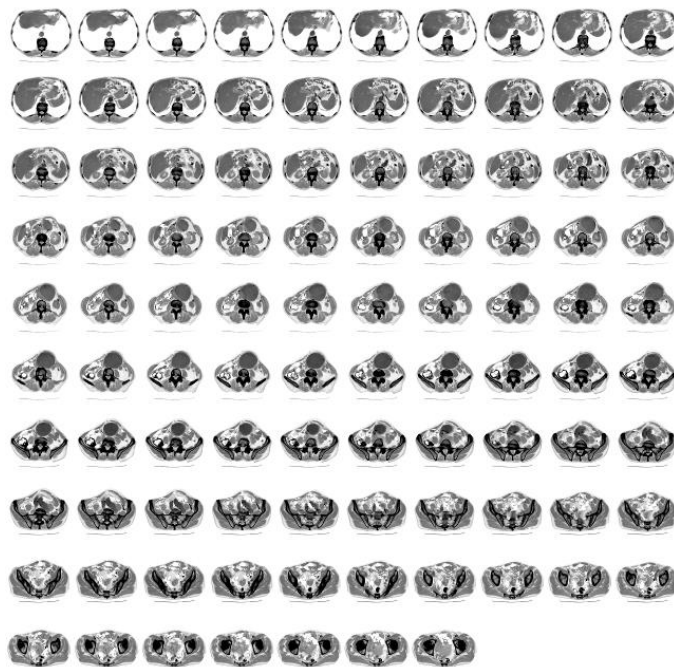


Figure 18 Texture mosaic for an aorta volume dataset

The shader code to fetch a voxel from the mosaic texture is demonstrated in Algorithm 1 which first calculates a z-slice number and the next z-slice. Based on the z-values, x and y coordinates are calculated and the correct location is determined for the 2D slice. Finally, texel values are fetched from the textures and final voxel color is blended from the fetched texel values. Quite many operations are needed to access a single voxel values from the mosaic texture. Performance would be increased a lot if 3D textures were supported in the WebGL API.

---

#### Algorithm 1 Fetching volume data from a mosaic texture

---

```

1: float getVolumeValue(vec3 volpos)
2: {
3:     float s1,s2;
4:     float dx1,dy1;
5:     float dx2,dy2;
6:     vec2 texpos1,texpos2;
7:     s1 = floor(volpos.z*numberOfSlices);
8:     s2 = s1+1.0;
9:     dx1 = fract(s1/slicesOverX);
10:    dy1 = floor(s1/slicesOverY)/slicesOverY;
11:    dx2 = fract(s2/slicesOverX);
12:    dy2 = floor(s2/slicesOverY)/slicesOverY;
13:    tp1.x = dx1+(volpos.x/slicesOverX);
14:    tp1.y = dy1+(volpos.y/slicesOverY);
15:    tp2.x = dx2+(volpos.x/slicesOverX);
16:    tp2.y = dy2+(volpos.y/slicesOverY);
17:    return mix( texture2D(uVol,tp1).x, texture2D(uVol,tp2).x, (volpos.z * numberOfSlices)-s1);
18: }

```

---

Performance of the ray casting algorithm is measured on desktop NVIDIA Geforce GTX 460 GPU in the 800 x 800 resolution. Some variance was found between the different browsers tested and the best results were achieved with Opera browser. Performance with high quality (200 steps) was 44 fps and with low quality (80 steps) 104 fps. Performance was also measured for mobile devices. However, only 2-3 fps was achieved on Samsung Galaxy Tab and Galaxy S mobile devices.

### 6.6. High-performance volume rendering on the ubiquitous webgl platform

A single-pass ray casting algorithm for WebGL is proposed by Mobeen et al (34). In single-pass ray casting box intersection math is used to determine if a ray hits the volume dataset or not. Therefore, one render pass is saved as there is no need to render cube back faces to separate an off-screen texture. Once valid ray is found, the volume dataset is ray casted in a normal way using a predetermined amount of steps.

The implementation supports transfer function editing. Different composition modes like MIP, maximum intensity difference accumulation (MIDA) and average composite are supported. Improved quality is achieved by using isosurface rendering. An isosurface can be determined so that a voxels that are inside the isosurface are rendered with opaque alpha values and voxels that are outside the isosurface are rendered using fully transparent voxels. Therefore, an isosurface is the surface between opaque and transparent voxels. Similarly, a three dimensional mathematical function that produces negative and positive values could form an isosurface for the values that are

exactly zero. Isosurfaces can be rendered using adaptive refinement. An adaptive process works with two voxel samples: one is inside the volume and the other is outside the volume. Once these values are found, an iterative process can begin where a new exact surface is searched for the inside and outside values. Adaptive refinement improvement is illustrated in Figure 19.

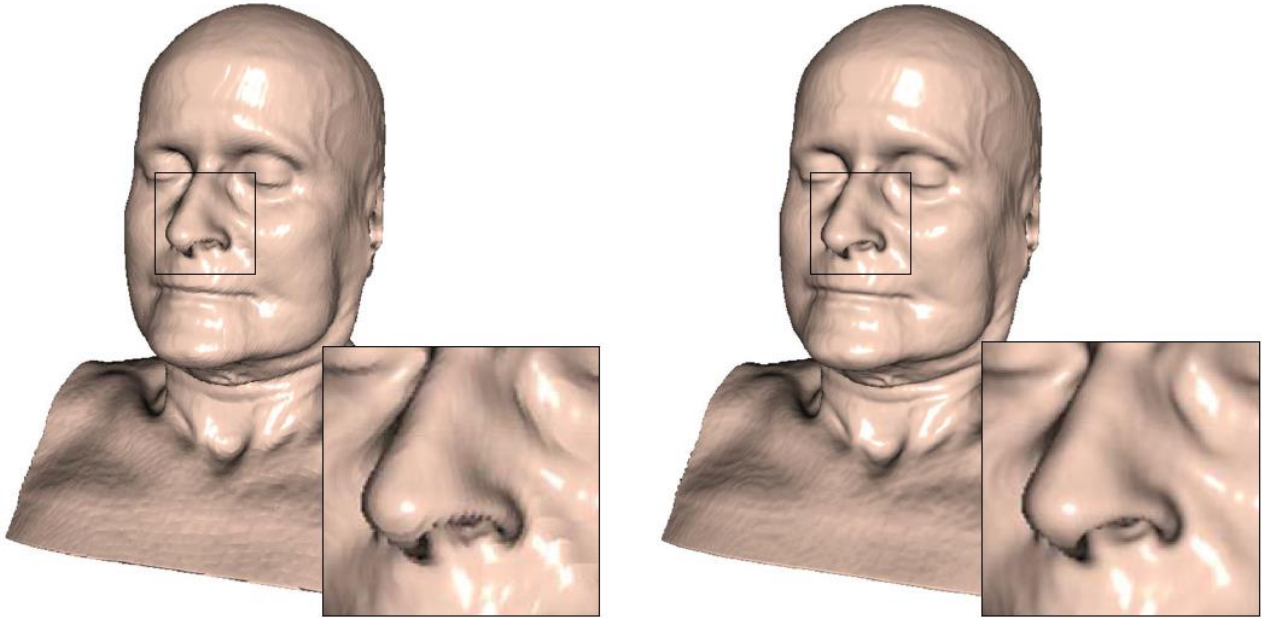


Figure 19 Quality improvement with adaptive refinement for iso-surfaces

Implementation supports BRDF (bi-directional reflectance distribution) shading. Shading improves the visual outlook of the volume by introducing shadow like areas and highlights.

Ray casting is benchmarked on mobile and desktop devices. Mobile devices include Acer Iconia 500 (NVIDIA Tegra 2 GPU) and Samsung Galaxy S II (Mali-400MP GPU) devices. Desktop rendering is tested on NVIDIA Quadro 5800 and Geforce 260M GPUs.

The proposed single-pass ray casting algorithm is compared with the multi-pass ray casting algorithm. For the 512 x 512 resolution results indicate that the single-pass ray casting is 2x faster than multi-pass ray casting for mobile devices. However, on desktop devices the difference between the single-pass and multi-pass is 1.1x – 1.7x.

### 6.7. Visualization of very large 3D volumes on mobile devices and WebGL

Typically the maximum size for a mosaic texture on mobile devices is limited to 2048 x 2048. This restricts the volume dataset to 128 x 128 x 256. However, Noguera et al (35) have proposed that extra color channels of textures and more texture units could allow larger volume datasets on mobile devices.



Normally a texture mosaic uses one 8-bit color channel for storing the volume slices. However, it is possible to use the full 32-bit channels and get 4 times more storage space for the volume slices. This can be achieved by adding extra selection to voxel fetching where a color channel is determined in the fetching process.

Similarly more texture units can be used to store more volume slices. OpenGL ES 2.0 specification allows 8 texture units to be used in the pixel shader. Therefore, voxel fetching can be modified to support fetching from different texture unit in the fetching process. Using full color channels of the texture mosaic and additional texturing units combined allows 32 times more storage for the mosaic texture. This technique allows large 512 x 512 x 512 volume datasets to be used with the limited 2048 x 2048 texture size.

Naturally, additional fetching logic to the shader code slows down the rendering performance. Performance between single and multi-channel mosaic textures is not benchmarked. However, final performance for the ray casting algorithm is provided for a mobile device and a desktop computer.

Rendering results include iPad 2 with 320 x 480 viewport size and NVIDIA Geforce 8800 GS in 800 x 800 resolution. It is possible to achieve 1.2 fps rendering speed with 80 steps for a 512 x 512 x 384 dataset on iPad 2. Clearly, the result is quite slow and an interactive frame rate is not achieved. Even the desktop computer with 128 steps for 512 x 512 x 1024 volume data sets only reaches 7.1 fps rendering speed.

## **6.8. Research summary**

It seems to be that most of the existing implementations use ray casting with mosaic textures. It is unfortunate that 3D textures are not in the core OpenGL ES 2.0 and WebGL specifications. Clearly, emulating 3D textures with mosaic 2D textures is slowing down the volume rendering on mobile devices. Unfortunately none of the presented papers compared 3D texturing to mosaic textures and therefore it is difficult to estimate how much mosaic texture access slows down the ray casting algorithm.

Most of the results that compared ray casting to slicing indicated that slicing was a faster way to render volumes. However, no deeper analysis was made in the papers what the core reason for this could be. As different device GPU architectures affect the benchmark results a lot it was a shame that most of the papers only used one or two devices in the benchmarking. Just using few devices in the benchmarking does not necessarily give the overall picture about the performance.

It was surprising that most of the papers did not include any optimization techniques like early exit or empty space skipping in their implementations as those are relatively simple to implement at least for the ray casting algorithm. Similarly, local lighting was not widely implemented.

## 7. Evaluation

Implementing a volume rendering engine for multiple platforms can be challenging. However, carefully selecting graphics APIs and programming languages can help porting the application to multiple platforms. Currently Apple's iOS and Google's Android system dominate the market share on mobile devices. Just by writing an application to these platforms will get a good coverage for testing volume rendering algorithms on mobile devices.

The target for the volume rendering engine was to maximize code reuse between the different supported platforms. The other target was to maximize the productivity of the engine development. This was achieved by writing the engine using Microsoft's Visual Studio and using OpenGL ES 2.0 emulation drivers. This allows to fully develop, debug and test the volume rendering algorithms first on the desktop and then compile the project for mobile devices. Usually a mobile development iteration cycle is much slower than a desktop development cycle as the application must be installed on the target device and this takes time.

There are multiple API options for developers targeting mobile and desktop devices. In order to fully create cross-platform application some evaluation is needed how to select proper APIs. Selecting platform specific APIs would lead to situation where most parts of the software need to be rewritten for the new platform. Supported graphics APIs and programming languages on each platform are illustrated in Table 1.

	<i>Windows</i>	<i>iOS</i>	<i>Android</i>	<i>HTML5</i>
<i>Java</i>			X	
<i>JavaScript</i>				X
<i>C / C++</i>	X	X	X (NDK)	X (Emscripten)
<i>Objective-C</i>		X		
<i>Swift</i>		X		
<i>CUDA</i>	X (NVIDIA)		X (NVIDIA K1)	
<i>RenderScript</i>			X (Android 4.2)	
<i>WebCL</i>				X (limited)
<i>OpenCL</i>	X		X (some devices)	
<i>Metal</i>		X		
<i>C++ AMP</i>	X			
<i>DirectCompute</i>	X			
<i>Direct3D</i>	X			
<i>OpenGL 4.5</i>	X		X (NVIDIA K1)	
<i>OpenGL ES 2.0</i>	X (emulation)	X	X	X (WebGL)

<b>OpenGL ES 3.0</b>	X (emulation)	X (iOS 7)	X (Android 4.3)	X (WebGL2)
<b>OpenGL ES 3.1</b>	X (emulation)		X (Android 5.0)	
<b>WebGL</b>				X
<b>WebGL2</b>				X

Table 1 API coverage on multiple platforms

APIs and the programming languages for the volume rendering engine should be selected so that the core engine is fully portable between the different platforms.

Programming language options for the engine development are Swift, Objective-C, C/C++, Java and JavaScript. Only C/C++ language is supported on all the target platforms except on web browsers. However, Emscripten is a compiler that can compile C/C++ code to JavaScript (36). This allows near native speed C/C++ code execution on web browsers.

Device hardware accelerators like the GPUs should be used to accelerate the volume rendering engine in order to fully utilize the device capabilities. Typically computer graphics algorithms use graphics programming APIs like OpenGL. However, as volume rendering is a highly parallel problem, other APIs could be used. Typically parallel programming APIs like CUDA, RenderScript, OpenCL, WebCL, C++ AMP and DirectCompute could be used. These APIs should give more programming access to the hardware. However, as seen from Table 1, parallel programming APIs are highly fragmented and not available on all the targeted platforms. Therefore, graphics programming API is selected to accelerate the volume rendering algorithms.

Multiple graphics programming APIs exist and the most popular APIs are Direct3D, OpenGL 4.5, different OpenGL ES versions and WebGL. Some proprietary APIs like Metal for iOS exist but as those APIs cannot be used on all platforms and therefore those are not considered for the volume rendering engine. As the volume rendering engine should run on iOS, Android and Web only OpenGL ES APIs can be considered. Only OpenGL ES version that runs on all targeted platforms is OpenGL ES 2.0 and the web version WebGL of it. Therefore, OpenGL ES 2.0 is selected for the volume rendering engine. However, the core OpenGL ES 2.0 API lacks the 3D texture support and this highly limits its full potential for volume rendering. Therefore, OpenGL ES 3.0 is also selected for the volume rendering engine. Benefit of the OpenGL ES 3.0 API is that 3D texture support is mandated by the OpenGL ES 3.0 specification. As OpenGL ES 3.1 is not supported on the latest iOS devices it was not considered for the volume engine development.

Volume engine is designed so that both OpenGL ES 2.0 and 3.0 APIs can be used. OpenGL ES 2.0 provides the full device coverage and OpenGL ES 3.0 API provides the latest features of the GPUs. In order to avoid using the mosaic textures in the volume rendering engine texture slicing is implemented for the OpenGL ES 2.0 API. This version of the volume rendering engine is provided

with limited functionality and it does not support for example local lighting and the optimization techniques. Full features are supported on the OpenGL ES 3.0 version of the volume engine and the ray casting algorithm is implemented. Ray casting suits relatively well for OpenGL ES 3.0 as it supports 3D textures and better shading language than OpenGL ES 2.0.

In order to build a cross-platform volume rendering engine the different parts of the application are split into different modules. Modules are illustrated in Figure 20. The core idea of the modules is that the application code and the rendering engine should not use any platform specific APIs in order to keep the portability of the application on good level. However, a separate utility module is provided for each platform with OS specific adaptation code. This part of the engine changed per platform and if new platform support is needed then only this part should be added to the engine.

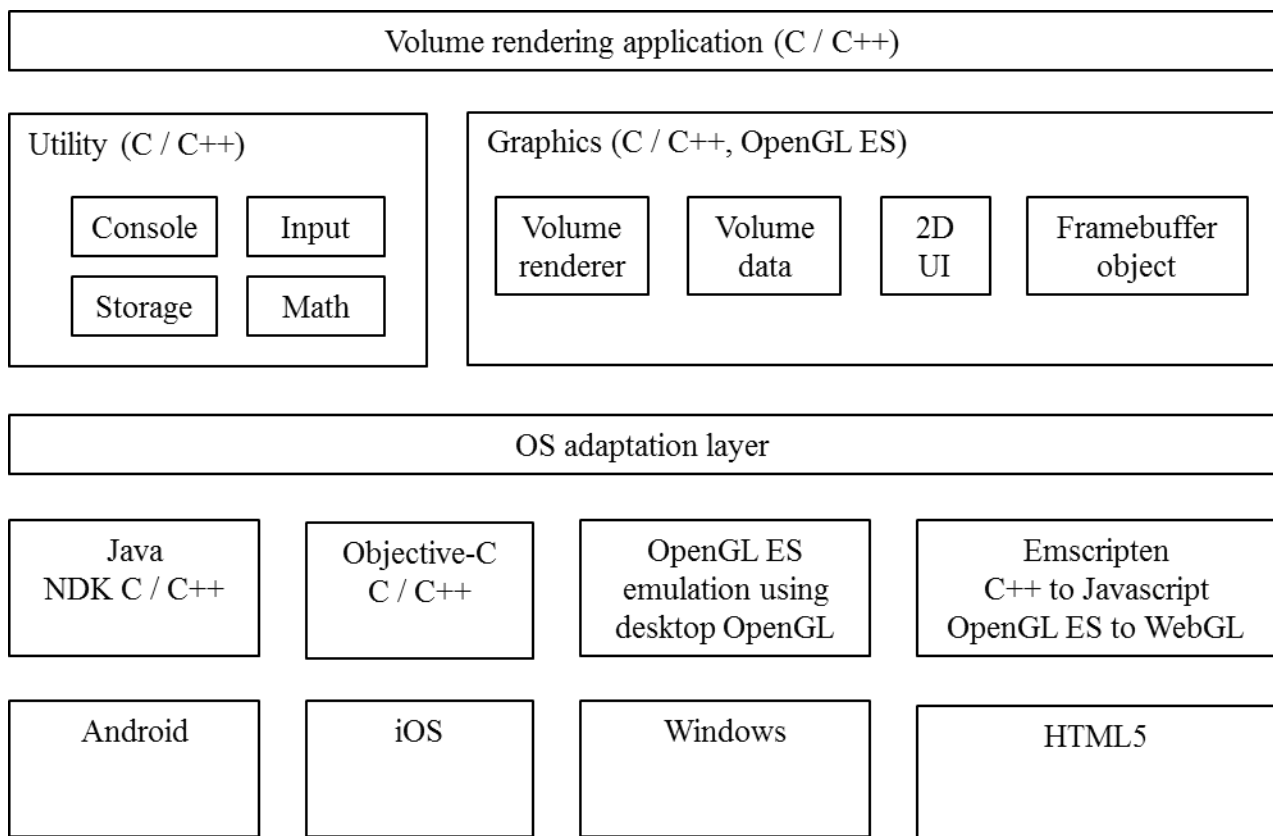


Figure 20 Software architecture for the volume rendering application

### 7.1. GPU hardware specifications

In order to evaluate performance on multiple GPUs it is good to understand the basic differences between GPU hardware features. Information about the GPUs was collected from different hardware reviews (37), (38), (39), (40) and (41). Typically most of the hardware vendors do not reveal their hardware features. This fact should be kept in mind when monitoring the different hardware features from Table 2 that there could be some errors. However, the table should give a rough estimate about the differences between the GPUs.

	<b>NVIDIA K1 (Nexus 9)</b>	<b>NVIDIA X1 (Dev. Board)</b>	<b>ARM Mali-T760 MP8 (Galaxy S6)</b>	<b>Imagination GXA6850 (iPad Air 2)</b>	<b>Qualcomm Adreno 420 (Nexus 6)</b>	<b>NVIDIA GTX 650 (Desktop)</b>
<b>FP32 ALUs</b>	192	256	80	256	?	384
<b>FP32 FLOPs</b>	384	512	160	512	?	768
<b>Pixels / Clock</b>	4	16	8	16	?	16
<b>Texels / Clock</b>	8	16	8	16	?	32
<b>GPU MHz</b>	852	1000	772	450	?	1000
<b>GFLOPS</b>	327	512	124	230	?	768
<b>ALU (fps)</b>	259	455	103	186	141	692
<b>Blending (MB/s)</b>	4045	21888	6772	17207	10427	21119
<b>Fill (MTexels/s)</b>	5632	12197	5169	7593	7326	20458

Table 2 GFXBench result and hardware specifications summary

GFXBench (1) is a popular multiplatform graphics performance benchmarking software that provides graphics performance results for several different graphics test content. Hardware features seem to match with the GFXBench results in Table 2.

Arithmetic operations (ALU) that GPU hardware can do depends on the number of processing cores. Speed increase in the GFXBench ALU test case can be seen when comparing NVIDIA K1 and X1 GPUs. This speed increase is mostly due to the fact that GPU processing cores have increased from 192 to 256 and also increase in GPU clock speed from 852 MHz to 1 GHz. There is direct relationship between the Pixels / Clock figure and the GFXBench blending rate. Pixels / clock figure means how many pixels can be rasterized in one GPU clock cycle. Pixels / clock figure can be increased by introducing new raster operations pipelines (ROPs) to the GPU hardware. Each ROP allows additional blending in a clock cycle. Increasing ROP count from 4 (K1) to 16 (X1) also increased blending rate with the same ratio.

A similar relationship exists for texels / clock and the GFXBench Fill test. Texels / clock provides the rate how many texture texels can be fetched in a GPU clock cycle. Each texture mapping unit (TMU) in the GPU increases the texels / clock figure. Volume rendering performance can be limited by ROP or TMU. Especially texture slicing can slow down dramatically if ROP count of the hardware is very low as the slicing is blending each voxel to the display buffer. As ray casting loop is running inside the pixel shader and outputs only one pixel to the display per ray, ROP count should not be so dominant performance limiter. However, both texture slicing and ray casting benefit from high TMU count as huge amount of texels are fetched from textures in volume

rendering. High number of texel fetches increases the texture bandwidth and when the system cannot provide more bandwidth performance decreases. In a case where high amount of textures are fetched in a rendered frame performance can be limited if there is not enough memory bandwidth from the texture memory. Performance is limited as texture cache of the GPU waits for new data and the pixel shader cannot continue its execution until the new texel data is ready.

While the texture cache stalls on waiting for a new texel data there is an opportunity for the GPU to do ALU calculations. In the ray casting algorithm composition is done in the pixel shader loop and therefore it could run parallel with the texture fetching. Similarly lighting calculations could be done parallel with the texture fetching. Level of parallelism depends on the GPU architecture.

## **7.2. Test setup**

Many different devices should be selected for testing the performance as there can be a wide variety of devices with different performance characteristics. However, with the limited budget it is not feasible to get tens of devices for performance testing. Devices were selected so that multiple GPU architectures are presented. Typically many of the devices are just copy products of each other with the same CPU and GPU combinations. A major part of the volume rendering algorithms run on the GPUs and therefore CPU has little effect on the final performance. Therefore, selecting devices with different GPUs is enough in order to get good performance coverage.

Devices can have variety of display resolutions. In order to compare different GPU architectures, volume rendering should be tested with the same display resolution. This can be achieved with off-screen textures. Once rendering is done for off-screen texture it can be displayed to the display by scaling the off-screen texture to match the display size. Benchmarked volume rendering test cases use the 512 x 512 off-screen texture. In benchmark mode rendering is done four times to the off-screen texture and each time small version is copied to the display in order to flush the GPU pipeline. This allows to reach 240 fps on devices that have displays synced to 60 fps display screen refresh rate.

Parallel projection is used as texture slicing and ray casting could provide different results in the rendering with perspective projection. In parallel projection far and near objects are rendered with the same size. This way the end result is basically a stack of 2D images for texture slicing that are exactly of the same size and drawn in back to front order. Texture slicing uses 512 pieces of slices on the top of each other. Ray casting on the other hand uses 512 steps in the pixel shader loop and outputs one colored pixel to the display per ray. Ray casting is done from front to back order. The same image is produced with texture slicing and ray casting. Therefore, it is possible to compare the performance of texture slicing with ray casting as they produce the same output with different algorithms.

Benchmarking results include optimization techniques that depend on the volume dataset content and also on the selected transfer function. Three different volume dataset are used from The Volume Library (42) in 8-bit 256 x 256 x 256 resolution. All the benchmark test cases produce frames per second (fps) number.

### 7.3. Slicing vs. ray casting

Performance difference between the ray casting and the texture slicing can be evaluated by fetching exactly the same amount of voxels and rendering the same number of pixels to the display. Performance difference between these two algorithms on a multiple mobile device GPUs can be seen in Table 3.

<i>Device</i>	<i>GPU</i>	<i>Ray casting</i>			<i>Slicing</i>	
		<i>Volume calculated (fps)</i>	<i>Small volume (fps)</i>	<i>Full volume (fps)</i>	<i>Small volume (fps)</i>	<i>Full volume (fps)</i>
Apple iPad 3	PowerVR SGX 543				7.0	6.8
Apple iPhone 5	PowerVR SGX 543				7.1	6.8
Apple iPad Air	PowerVR G6430	13.7	6.6	6.6	11.9	11.4
Apple iPad Air 2	PowerVR GXA6850	39.2	16.0	16.0	28.5	27.3
Galaxy S4 (GT-I9295)	Adreno 320				10.2	9.1
Nexus 5	Adreno 330	10.7	12.8	3.4	11.9	11.2
Galaxy S5 (SM-G900F)	Adreno 330	18.3	14.4	3.5	14.2	12.9
Nexus 6	Adreno 420	16.0	18.3	4.3	18.4	11.7
Nexus 9	NVIDIA Tegra K1	36.3	22.2	19.6	21.2	14.9
Galaxy S6 (SM-G920F)	Mali-T760	14.5	20.8	6.1	11.8	5.9

Table 3 Performance difference between the slicing and the ray casting algorithms

Benchmarked results include Volume calculated, Small volume and Full volume test cases. In the Volume calculated test case no voxels are fetched but the geometry of the volume dataset is



emulated by procedurally generating sphere geometry with mathematical functions. This test case should indicate the amount of ALU performance of the GPU and can be used to estimate if the texture bandwidth or fetching is the limiting factor for performance. Results indicate that on GXA6850 and Tegra K1 GPU architectures Volume calculated test case provides very high performance compared to fetching actual voxels. Both of these GPU architectures include a high number of ALU units that can explain this behavior.

Small volume test case fetches all the voxels from the volume dataset but the volume dataset is decreased to the 16 x 16 x 16 size. However, the same number of voxels are processed as in the Full volume test case. Therefore, the main difference between the Small and Full test cases is the amount of texture bandwidth generated by the test cases. A major drop in performance can be seen when comparing Small and Full test cases for Adreno and Mali based GPUs for ray casting. This clearly indicates that these GPU architectures are not well suited for fetching data using 3D textures. However, on the Adreno GPU the texture slicing performance is not decreased with the same factor as with the ray casting. This indicates that the problem is 3D texture fetching related as the texture slicing is fetching 2D textures.

Results indicate that the texture bandwidth is not the problem for GXA6850 and Tegra K1 architectures as the Small and the Full volume test cases are almost equal in performance. As both of these GPU architectures performed well on the Calculated volume test case it means that texture fetching is somewhat limited on these GPU architectures. In practice TMU and ROP are limiting factors not the texture bandwidth. Therefore, performance could only be increased when increasing the number of TMU and ROP hardware units.

Results indicate that the slicing is faster than the ray casting with the full volume dataset. This can be explained with the fact that slicing uses the dedicated ROP hardware for the blending which can operate parallel with the pixel shader code. However, in the ray casting blending is done in the shader code and this decreases the performance as the ROP hardware cannot be utilized at all. Performance is also decreased in the ray casting test case due to the fact that it is using 3D texture fetching and not 2D textures.

Quality is an important factor in the volume rendering. Typically more quality is achieved when putting more resources to the output calculation. In volume rendering quality can be simply increased by adding more iterative steps in the ray casting algorithm or adding more layers to the texture slicing algorithm. At least the same number of steps or layers should be used as in the volume dataset. Therefore, 256 steps or layers are required if the volume dataset is 256 x 256 x 256. A problem with the texture slicing algorithm is the rotated proxy geometry. As the volume is rotated close to the 45 degree angles, gaps between the layers can be observed if there are not enough layers in the texture slicing. Similarly the ray casting algorithm could skip some of the

voxels if step count is too low. This typically happens when the ray is casted from the corner of the cube to the opposite corner of the cube. In this case 256 steps are not enough if all the voxels must be traversed for the  $256 \times 256 \times 256$  dataset. Therefore, in both cases to avoid most of the aliasing artifacts a greater sampling rate is needed in order to render good quality volume rendered images. A high quality step count of 512 was selected for all the benchmarked test cases. Quality difference between lower the numbers of steps can be observed from the Figure 21. Clearly poor results are achieved with 64 steps and good results can be achieved with the full 512 steps.



Figure 21 Rendering quality difference between ray casting (upper row) and slicing (bottom row) with 64 (left), 128, 256 and 512 (right) steps.

The transfer function is a key functionality in the volume rendering. With the transfer function different parts of the volume can be colored with different colors. Similarly parts of the volume can be made fully transparent or even fully opaque. As an example the densest parts of the volume could be set to opaque and the low density parts to fully transparent by editing the transfer function. Typically, if the volume dataset voxels are 8-bit values, a 256-element transfer function lookup table is needed. In the transfer function each volume dataset value is mapped to a 32-bit RGBA color value.

The transfer function can be pre-calculated and in this case 32-bit RGBA values would be used for the volume dataset. In the benchmark test cases Full volume test case uses this approach. A disadvantage in the pre-calculated transfer function is that whenever the transfer function is changed also the volume dataset needs to be changed. This requires some processing. However, the processing is only done once when the transfer function changed and not during each rendered frame. In the case where a lookup table is used a volume dataset is fetched and another a fetch from the lookup table is needed.

Typically the lookup table is stored to a 256-element 1D texture and can be easily used in the shader code. However, another option would be to use pixel shader constants to store the lookup table data. In this option no additional textures would be fetched and this could provide more performance.

<i>Device</i>	<i>GPU</i>	<i>Ray casting</i>			<i>Slicing</i>		
		<i>Full volume (fps)</i>	<i>Palette transfer function (fps)</i>	<i>Texture transfer function (fps)</i>	<i>Full volume (fps)</i>	<i>Palette transfer function (fps)</i>	<i>Texture transfer function (fps)</i>
Apple iPad 3	PowerVR SGX 543				6.8		4.4
Apple iPhone 5	PowerVR SGX 543				6.8		4.5
Apple iPad Air	PowerVR G6430	6.6	6.6	5.3	11.4	5.6	7.6
Apple iPad Air 2	PowerVR GXA6850	16.0	15.9	12.8	27.3	16.1	19.1
Galaxy S4 (GT-I9295)	Adreno 320				9.1		6.8
Nexus 5	Adreno 330	3.4	2.8	5.2	11.2	5.5	8.4
Galaxy S5 (SM-G900F)	Adreno 330	3.5	3.9	7.3	12.9	7.0	10.5
Nexus 6	Adreno 420	4.3	4.5	6.7	11.7	6.8	14.2
Nexus 9	NVIDIA Tegra K1	19.6	16.5	13.4	14.9	11.5	10.7
Galaxy S6 (SM-G920F)	Mali-T760	6.1	13.1	9.7	5.9	5.7	6.2

Table 4 Performance difference between the palette and the texture transfer functions.

Benchmarking results in Table 4 include Full volume, Palette transfer function and Texture transfer function test cases. Full volume test case uses the pre-calculated transfer function and storage size is 32-bit RGBA values per voxel. Both transfer function based test cases use lookup tables with 8-bit voxel storage. Therefore, the texture bandwidth is high with the Full volume test cases and 4 times lower with the transfer function test cases. In the Palette transfer function the lookup table

data is stored to shader constants and in the Texture transfer function case lookup table is stored to a new texture. Limitation with the Palette test case is that only few GPU hardware architectures support a high number of constants. This means that the full 256-element palette cannot fit for the shader constants. A compromise was made and a 60-element reduced palette lookup table was used in order to benchmark the difference between the Palette and Texture based transfer functions.

Results indicate that generally the texture based transfer function works faster than the palette based transfer function for the texture slicing algorithm. Even though the palette based fetching does not read textures it requires some logic to determine the lookup table index. This causes some extra calculations in the shader code. However, as the palette fetching is faster on the ray casting for the GXA6850 and for the Tegra K1 GPUs it means that if there is enough ALU power in the GPU the palette case works better. Typically reading a transfer function slows down the rendering with the slicing algorithm. Therefore, it would be better to use the pre-calculated transfer function if volume rendering was to be implemented with the slicing algorithm. With the ray casting algorithm a clear choice is to use the texture based transfer function with the Adreno GPUs. On the other hand, Mali bases GPUs work better with the palette based transfer function.

Even though the texture based lookup table is not the fastest one for all the GPUs it was selected as a basis for the rest of the benchmarking test cases as it provides the most flexibility and consumes less memory.

#### 7.4. Local illumination

Illumination can add more realism to the volume rendering and should be implemented for all volume renderers. However, adding any illumination model to the volume rendering requires normals to be calculated for the voxels. A voxel normal can be either a pre-calculated or a calculated on the fly.

In the pre-calculated case volume normals are calculated in the loading phase and a high number of operations can be applied to the calculations. However, a new normal volume dataset needs to be stored and typically this takes 24-bits per voxel more memory to store the normals as a volume. High quality Sobel based normals were calculated as a pre-calculated step.

On the fly calculated normals take the neighbor voxel values and normals are calculated from the differences. Forward difference uses the current voxel and three other voxels to calculate the normals. Central difference improves this by even fetching more voxels to determine the final normals.

		<i>No illumination</i>	<i>Blinn-Phong illumination</i>
--	--	------------------------	---------------------------------

<i>Device</i>	<i>GPU</i>	<i>Texture transfer function (fps)</i>	<i>Forward gradient normals (fps)</i>	<i>Central gradient normals (fps)</i>	<i>Sobel 3D filtered normals (fps)</i>
Apple iPad Air	PowerVR G6430	5.3	1.5	2.5	3.0
Apple iPad Air 2	PowerVR GXA6850	12.8	3.8	2.2	6.5
Nexus 5	Adreno 330	5.2	2.1	1.2	1.6
Galaxy S5 (SM-G900F)	Adreno 330	7.3	2.8	1.6	1.7
Nexus 6	Adreno 420	6.7	2.5	1.8	1.9
Nexus 9	NVIDIA Tegra K1	13.4	4.6	2.7	7.7
Galaxy S6 (SM-G920F)	Mali-T760	9.7	3.2	1.8	3.2

Table 5 Performance difference on the fly and pre-calculated normal generation

Results for the on the fly and pre-calculated normals are illustrated in Table 5. As a comparison texture based transfer function results are included in the results. Therefore, any extra calculations will decrease the performance results from the texture based transfer function that includes no light calculations at all. Illumination is only implemented for the ray casting algorithm.

Performance is decreased by the factor of 2 in most of the cases when the illumination is calculated for all the voxels. Therefore, it seems that the illumination requires quite heavy processing. As previously noted, the fact that Adreno GPUs seem to have problems with 3D texture fetching results indicate that normals should be calculated using the forward difference method for Adreno based GPUs. As pre-calculated normals require more bandwidth and the central difference normals more texture fetches, these options are not well suited for Adreno based GPUs. For the rest of the GPUs where bandwidth is not the issue compared to the additional texture fetching, pre-calculated normals offer the best performance. As the pre-calculated normals offer the best performance on the most high-end devices and also the best quality it was selected as the basis for the optimization techniques testing.

## 7.5. Optimization techniques

Multiple optimization techniques exist for volume rendering. Most typical optimization techniques try to avoid processing empty areas or fully opaque areas of the volume rendering. Other techniques exist to improve memory management of the volume dataset via compression or better texture cache behavior.

Benchmarked optimization techniques mainly focus on improving the user perceived performance (fps) by reducing volume calculations. Previous illumination test cases calculated shading for each voxel. However, there is no need to calculate the shading if voxel is fully transparent. Therefore, some lighting calculations could be avoided for some areas of the volume. Three different volume datasets are benchmarked as the dataset affects how many voxels are fully transparent.

All the tested optimization techniques are implemented for the ray casting algorithm. Some of tested algorithms would work also with the texture slicing but would require a different approach. In texture slicing each slice is rendered separately and the only way to avoid calculations per slice would be to split the slice into smaller pieces. Slices could be for example split into 16 x 16 pixel quads and for each quad a test would be performed if the quad is fully transparent or not. Quads that are fully transparent would not need to be rendered. However, the only way to split the slice into quads would be to create new geometry out of the generated quads. This would introduce new vertices for the slice rendering and depending on how many vertices are generated it could also affect the performance. Splitting slices into quads would reduce light calculations and also unnecessary drawing of fully transparent pixels.

The simplest way to accelerated empty area skipping for ray casting would be to use distance volume. Distance volume would be a new volume where each voxel element would contain the shortest distance to the semi-transparent or opaque surface. Distance volume would allow a ray casting loop to skip iterations based on the distance value. However, a disadvantage of using the distance volume is that it needs to be recalculated each time when the transfer function is changed as new areas of the volume could become fully transparent. This would require a heavy pre-processing step. Another disadvantage of using distance volumes is the additional memory usage. Distance volume would consume up to four times more memory than the original volume dataset if floating point distance values were used. As memory bandwidth is limited on the mobile devices some other ways to skip the empty space areas would be preferred.

A simple lookup table for empty areas was used to optimize empty area skipping. The size of the lookup table is 16 x 16 x 16 and each element of the lookup table indicates if that specific block can be fully skipped or not. Therefore, one single 16 x 16 x 16 block can be skipped at time if empty

blocks are found. Calculation of the lookup table is simple. In the first pass all the volume voxels are checked if they contain fully transparent values or not. If a block is fully transparent, it is marked that it can be skipped. However, a second pass is needed for the reduced blocks as skipping cannot happen unless all the neighbor blocks are also transparent. Once this step is completed the low resolution 3D lookup table contains values that can be used in the ray casting algorithm to determine if several ray casting steps can be skipped or not. This algorithm could be improved towards the distance volume algorithm so that distances to the surfaces would be calculated in a low resolution and therefore the algorithm might run with acceptable speed. A benefit of this low resolution lookup skipping table is that it consumes a low amount of memory and is faster to update than the full distance volume. However, it is equally optimal solution as the full distance volume solution.

The other optimization technique for ray casting is to exit the ray casting loop when opaque voxels are found. As the ray casting is done in front to back order any voxels behind an opaque voxel can be discarded as the end result would not be affected. An early exit does not help the performance if all the voxels are fully or partially transparent. Therefore, the volume dataset should contain some amount of opaque voxels so that the benefit of early exit would show.

<i>Device</i>	<i>GPU</i>	<i>Engine dataset</i>				<i>Head dataset</i>				<i>Foot dataset</i>			
		<i>Process all voxels (fps)</i>	<i>Skip light (fps)</i>	<i>Early exit (fps)</i>	<i>Skip empty space (fps)</i>	<i>Process all voxels (fps)</i>	<i>Skip light (fps)</i>	<i>Early exit (fps)</i>	<i>Skip empty space (fps)</i>	<i>Process all voxels (fps)</i>	<i>Skip light (fps)</i>	<i>Early exit (fps)</i>	<i>Skip empty space (fps)</i>
Apple iPad Air	PowerVR G6430	3.0	4.9	5.5	9.1	3.0	4.0	6.4	6.6	2.9	4.8	5.6	7.4
Apple iPad Air 2	PowerVR GXA6850	6.6	11.9	13.4	21.7	6.6	9.7	16.0	15.9	6.4	11.6	13.9	17.8
Nexus 5	Adreno 330	2.2	4.6	4.8	7.1	1.9	2.7	4.9	4.8	0.8	1.7	2.0	3.2
Galaxy S5	Adreno 330	2.1	5.4	6.2	9.7	1.8	2.6	4.8	4.7	1.1	2.7	3.2	4.3

(SM-G900F)													
Nexus 6	Adreno 420	2.2	4.9	5.4	9.6	2.1	2.7	5.4	5.6	1.4	3.4	4.0	5.5
Nexus 9	NVIDIA Tegra K1	9.2	11.0	12.9	21.0	7.9	7.2	12.0	14.1	6.1	8.7	10.4	15.1
Galaxy S6 (SM-G920F)	Mali-T760	4.2	6.9	7.9	8.1	2.9	1.7	5.9	5.8	2.6	3.6	4.5	3.2

Table 6 Benchmarking results for optimization techniques

Results for the different optimization techniques are illustrated in Table 6. A baseline for the results is the lighting test case with pre-calculated normals. In this case all the voxels are processed with light calculations. Results include test case where the light calculations are discarded for fully transparent voxels. An early ray termination and empty space skipping are also included in the results.

Overall, skipping the light calculations can increase the performance over by a factor of 2. It seems that only in the head volume dataset performance is bit lower with light skipping on some of the GPUs. This could be due to the fact that there are not many fully transparent empty areas in the head dataset.

Results indicate that an early exit has a major effect on the head dataset. Performance is accelerated by almost factor of 2 on best cases. As the head dataset includes a lot of near fully opaque voxels ray casting can be exited. However, skipping empty space does not gain any performance for the head dataset. As skipping would require fully empty blocks there is no benefit for this specific dataset. Other datasets benefit from the empty space skipping and almost 2x gain in performance can be seen.

Datasets and the transfer function affect how well the optimization techniques work. However, performance can be greatly enhanced with optimization techniques as some of the techniques work for fully transparent voxels and some techniques for near opaque voxels. Based on the results using optimization techniques is highly recommend as using all the techniques combined resulted to higher level of performance. Some corner cases where the performance was decreased by the optimization technique could be detected and that specific technique could be disabled. Detection could be based on calculating fully transparent, semi-transparent and opaque areas of the volume dataset. Based on the results new shaders would be generated that do not contain the specific optimization technique that slows down the performance.



## 7.6. Native runtime vs. web runtime

As the developed rendering engine can be compiled to web environment without modifications it is possible to evaluate the native runtime performance versus the web runtime performance. Due to the fact that web browsers run on a sandbox environment and JavaScript includes lot of safety checks performance has been always lower with web runtime compared to native runtime. In native runtime where C/C++ programs are directly compiled to CPU processor instructions the full performance can be reached but no safety checks like buffer under and overflows are checked runtime.

Volume rendering is a highly parallel algorithm that runs fully on the GPU cores. Very little processing is done with the CPU. In practice this means that volume rendering is well suited for web runtime as there is little JavaScript code that need to be executed for each frame. Most of the performance critical code is in the graphics driver and the processing is happening in the GPU.

Actual volume rendering performance comparison for native and web runtime were tested on iPad Air 2 mobile device and with iMac desktop machine. Desktop performance was tested using Windows and MacOS operation systems on the same hardware. Windows can be run on the iMac using BootCamp. Native performance on the desktop test was tested with OpenGL ES 2.0 wrapper drivers that run on top of the real OpenGL drivers. The wrapper driver could introduce some performance penalty compared to directly using OpenGL drivers. Mobile testing was done with the created native mobile application and running the web version of the application on Safari browser.

Results in Table 7 are benchmarked with the latest browser versions that were available during the testing. On some Windows browsers WebGL API calls could be translated to Direct3D calls. On MacOS desktop OpenGL drivers are most likely to be used in the browser. Results clearly indicate that web version of the application runs at the same speed as the native application. The only exception is the Safari browser that has lower performance than the other browsers on MacOS.

		<i>Engine dataset</i>	<i>Head dataset</i>	<i>Foot dataset</i>
--	--	-----------------------	---------------------	---------------------

<i>OS</i>	<i>Web browser</i>	<i>Small volume (fps)</i>	<i>Full volume (fps)</i>	<i>Texture transfer function (fps)</i>	<i>Small volume (fps)</i>	<i>Full volume (fps)</i>	<i>Texture transfer function (fps)</i>	<i>Small volume (fps)</i>	<i>Full volume (fps)</i>	<i>Texture transfer function (fps)</i>
iOS	Native	28.1	27.3	19.4	28.7	27.2	18.7	28.7	27.3	19.3
iOS	Safari	28.7	27.3	19.0	28.7	27.0	18.7	28.5	27.1	19.2
Windows	Native	42.3	37.4	27.2	42.1	32.3	25.3	42.1	35.1	26.8
Windows	Firefox	42.0	36.8	25.0	41.9	31.9	23.4	41.9	34.6	24.3
Windows	Chrome	41.8	36.8	25.1	41.6	31.9	23.2	41.7	34.7	24.7
MacOS	Firefox	41.4	36.9	25.5	41.5	32.0	25.2	41.5	34.7	24.9
MacOS	Chrome	41.9	37.2	25.4	41.8	32.1	25.4	42.4	35.6	24.9
MacOS	Safari	34.3	33.8	23.8	34.5	29.4	22.5	34.3	30.0	21.8

Table 7 Performance difference between native and web based volume renderers

## 8. Conclusion

This thesis presented the theory behind the volume rendering by researching the basic algorithms created for rendering, illumination and optimization techniques.

Existing research literature for mobile and web volume renderers is covered and literature results are analyzed and used as a basis for implementing volume renderer for mobile devices. Problems with the results of the existing mobile volume renderers are discovered as most of the implementations only use a few devices in their performance evaluation. Therefore, the results can be too biased toward certain GPU architectures.

The contribution of this thesis is the extensive benchmarking results for multiple mobile GPU architectures. Several key volume rendering algorithm options are compared and evaluated and the key performance bottlenecks are identified. Results indicate that texture fetching and texture memory bandwidth are the most limiting factors for volume rendering performance on the mobile GPUs. High variance in the results is also found between different mobile GPU architectures. This is problematic and makes using the same set of algorithms challenging for all the mobile GPU architectures. When taking texture fetching and bandwidth problem on mobile devices into account, the need for algorithm optimizations is a key factor for achieving better performance. Some of the most widely used optimization techniques are tested on mobile devices and results indicate that a high performance increase can be achieved by the optimization algorithms. Finally, the performance of the native and web runtimes are compared. Results indicate that similar performance for native and web runtimes can be achieved for graphically intensive applications like volume rendering.

Future research could explore texture fetching in more detail by analyzing different texture formats, filtering modes and comparing mosaic, 2D and 3D texturing. Similarly, more emphasis could be put on analyzing texture bandwidth. New algorithms could be developed to reduce the bandwidth. Even though compressed textures are supported by all the mobile GPUs, they were not evaluated in this thesis. Evaluating the performance of compressed textures could be interesting as compression saves texture bandwidth. However, the quality and other limitations should carefully be evaluated. Some of the parallel programming APIs like CUDA are nowadays available on mobile devices. API features and a performance analysis should be done in order to evaluate the feasibility of the parallel programming APIs for volume rendering algorithms.

Finally, the upcoming WebGL2 API could be tested once standardization is finished. The advantage of WebGL2 API is that it supports 3D textures and volume rendering should gain a significant performance boost compared to the current WebGL standard.

## References

- (1) GFXBench benchmark database. 2015; Available at: <http://gfxbench.com>. Accessed 6/3, 2015.
- (2) 3D scan-conversion algorithms for voxel-based graphics. Proceedings of the 1986 workshop on Interactive 3D graphics: ACM; 1987.
- (3) Fuchs H, Kedem ZM, Uselton SP. Optimal surface reconstruction from planar contours. Commun ACM 1977;20(10):693-702.
- (4) Keppel E. Approximating complex surfaces by triangulation of contour lines. IBM Journal of Research and Development 1975;19(1):2-11.
- (5) Herman GT, Liu HK. Three-dimensional display of human organs from computed tomograms. Computer graphics and image processing 1979;9(1):1-21.
- (6) Marching cubes: A high resolution 3D surface construction algorithm. ACM siggraph computer graphics: ACM; 1987.
- (7) Shirley P, Tuchman A. A polygonal approximation to direct scalar volume rendering. : ACM; 1990.
- (8) Fast volume rendering using a shear-warp factorization of the viewing transformation. Proceedings of the 21st annual conference on Computer graphics and interactive techniques: ACM; 1994.
- (9) Interactive volume rendering. Proceedings of the 1989 Chapel Hill workshop on Volume visualization: ACM; 1989.
- (10) Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. Proceedings of the 1994 symposium on Volume visualization: ACM; 1994.
- (11) Wilson O, Van Gelder A, Wilhelms J. Direct Volume Rendering Via 3D Textures. : Citeseer; 1994.
- (12) Levoy M. Display of surfaces from volume data. Computer Graphics and Applications, IEEE 1988;8(3):29-37.
- (13) Acceleration techniques for GPU-based volume rendering. Proceedings of the 14th IEEE Visualization 2003 (VIS'03): IEEE Computer Society; 2003.
- (14) Compositing digital images. ACM Siggraph Computer Graphics: ACM; 1984.
- (15) Engel K, Hadwiger M, Kniss J, Rezk-Salama C, Weiskopf D. Real-time volume graphics. : CRC Press; 2006.
- (16) Sobel I. Camera models and machine perception 1970.

- (17) Phong BT. Illumination for computer generated pictures. Commun ACM 1975;18(6):311-317.
- (18) Models of light reflection for computer synthesized pictures. ACM SIGGRAPH Computer Graphics: ACM; 1977.
- (19) Casting curved shadows on curved surfaces. ACM Siggraph Computer Graphics: ACM; 1978.
- (20) Kniss J, Kindlmann G, Hansen C. Multidimensional transfer functions for interactive volume rendering. Visualization and Computer Graphics, IEEE Transactions on 2002;8(3):270-285.
- (21) Levoy M. Efficient ray tracing of volume data. ACM Transactions on Graphics (TOG) 1990;9(3):245-261.
- (22) Acceleration of ray-casting using 3-D distance transforms. Visualization in Biomedical Computing: International Society for Optics and Photonics; 1992.
- (23) Levoy M. Volume rendering by adaptive refinement. The Visual Computer 1990;6(1):2-7.
- (24) Efficiently rendering large volume data using texture mapping hardware. Data Visualization'99: Springer; 1999.
- (25) Blythe D. OpenGL ES Common / Common-Lite Profile Specification. 2004; Available at: [https://www.khronos.org/registry/gles/specs/1.0/opengles\\_spec\\_1\\_0.pdf](https://www.khronos.org/registry/gles/specs/1.0/opengles_spec_1_0.pdf). Accessed 5/31, 2015.
- (26) Interactive volume rendering on mobile devices. Vision, Modeling, and Visualization VMV; 2008.
- (27) Munshi A, Leech J. OpenGL ES Common Profile Specification Version 2.0.25. 2010; Available at: [https://www.khronos.org/registry/gles/specs/2.0/es\\_full\\_spec\\_2.0.25.pdf](https://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf). Accessed 5/31, 2015.
- (28) Rodríguez MB, Alcocer PPV. Practical Volume Rendering in Mobile Devices. Advances in Visual Computing: Springer; 2012. p. 708-718.
- (29) Volume Rendering Strategies on Mobile Devices. GRAPP/IVAPP; 2012.
- (30) Direct volume rendering with shading via three-dimensional textures. Volume Visualization, 1996. Proceedings., 1996 Symposium on: IEEE; 1996.
- (31) Interactive 3D Image Processing System for iPad. Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2012 4th International Conference on: IEEE; 2012.
- (32) Jackson D. WebGL specification. 2014; Available at: <https://www.khronos.org/registry/webgl/specs/1.0/>. Accessed 6/1, 2015.
- (33) Interactive visualization of volumetric data with webgl in real-time. Proceedings of the 16th International Conference on 3D Web Technology: ACM; 2011.

- (34) High-performance volume rendering on the ubiquitous webgl platform. High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICISS), 2012 IEEE 14th International Conference on: IEEE; 2012.
- (35) Noguera JM, Jiménez J. Visualization of very large 3D volumes on mobile devices and WebGL. 2012.
- (36) Emscripten. 2015; Available at: <http://kripken.github.io/emscripten-site/>. Accessed 6/3, 2015.
- (37) Humrick M. Nvidia Announces Maxwell - Powered Tegra X1 SoC at CES. 2015; Available at: <http://www.tomshardware.co.uk/nvidia-tegra-x1-soc-ces,news-49498.html>. Accessed 5/30, 2015.
- (38) High-performance volume rendering on the ubiquitous webgl platform. High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICISS), 2012 IEEE 14th International Conference on: IEEE; 2012.
- (39) Smith R. Apple A8X's GPU - GXA6850, Even Better Than I Thought. 2014; Available at: <http://www.anandtech.com/show/8716/apple-a8xs-gpu-gxa6850-even-better-than-i-thought>. Accessed 5/30, 2015.
- (40) Smith R. ARM's Mali Midgard Architecture Explored. 2014; Available at: <http://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored/8>. Accessed 5/30, 2015.
- (41) Smith R. Imagination's PowerVR Rogue Architecture Explored. 2015; Available at: <http://www.anandtech.com/show/7793/imaginations-powervr-rogue-architecture-exposed/5>. Accessed 5/30, 2015.
- (42) Volume datasets. 2015; Available at: <http://lgdv.cs.fau.de/External/vollib>. Accessed 6/3, 2015.

## Glossary

**Voxel** A single 3D data element of volume.

**Texel** A single 2D data element of a texture.

**Texture** A 2D dimensional set of pixels.

**Vertex** A point in a 3D space.

**Triangle** A basic rendering primitive that can be formed using three vertices.

**Shader** A computer program that shades pixels or calculates new vertex positions.

**Raster operations pipeline** Hardware unit that rasterizes and blends pixels to the display.

**z-buffer** A buffer where depth distance to the viewer is stored.

**Texture cache** Cache where texture samples are temporarily stored.

**Texture mapping unit** Hardware unit that fetches texels from memory and applies filtering.

**Texture mapping** Process where textures are rendered to a display.

**Illumination** Adds lighting calculations for pixels.

**Isosurface** A discrete representation of geometry.

**Polygon** A closed shape bounded by only straight lines.

**Multi-texturing** A texturing method where multiple textures are combined together.

**Texture coordinates** A set of 2D or 3D coordinates that define where texture texels are fetched.

**Composition** A method of overlaying color layers.

**Acronyms**

**2D** Two-dimensional.

**3D** Three-dimensional.

**ALU** Arithmetic logic unit.

**API** Application programming interface.

**BRDF** Bidirectional reflectance distribution function.

**CPU** Central processing unit.

**GPU** Graphics processing unit.

**MIDA** Maximum intensity difference accumulation

**MIP** Maximum intensity projection.

**ms** Millisecond.

**FP32** 32-bit floating point

**FLOP** Floating-point operations per second.

**GFLOP** gigaFLOPS.

**MHz** Megahertz.

**RGBA** Red, green, blue and alpha color channels.

**ROP** Raster operations pipeline.

**TMU** Texture mapping unit.

**OS** Operation system.



## Appendices

### Slicing shader programs

#### *Vertex shader*

```
attribute vec4 av4position;
attribute vec4 av3texcoord;

varying vec4 vv3Texcoord;

uniform mat4.mvp;

void main() {
    vv3Texcoord = av3texcoord;
    gl_Position =.mvp * av4position;
}
```

#### *Fragment shader*

```
precision mediump float;

uniform sampler2D.tflUT;
uniform vec4.paletteLUT[60];

uniform sampler2D.volumeSlicePrecalcSmall;
uniform sampler2D.volumeSlicePrecalc;
uniform sampler2D.volumeSlice;

uniform sampler2D.volumeSlicePrecalcSmallN;
uniform sampler2D.volumeSlicePrecalcN;
uniform sampler2D.volumeSliceN;

varying vec4 vv3Texcoord;
```

```

vec4 fetchVolume(vec3 pos)
{
#ifdef TF_PRECALC_SMALL
    vec4 color1 = texture2D(volumeSlicePrecalcSmall, pos.xy);
    vec4 color2 = texture2D(volumeSlicePrecalcSmallN, pos.xy);
    vec4 color = mix(color1, color2, pos.z);
#endif
#ifdef TF_PRECALC
    vec4 color1 = texture2D(volumeSlicePrecalc, pos.xy);
    vec4 color2 = texture2D(volumeSlicePrecalcN, pos.xy);
    vec4 color = mix(color1, color2, pos.z);
#endif
#ifdef TF_TEXTURE
    float voxel1 = texture2D(volumeSlice, pos.xy).r;
    float voxel2 = texture2D(volumeSliceN, pos.xy).r;
    float voxel = mix(voxel1, voxel2, pos.z);
    vec4 color = texture2D(tfLUT, vec2(voxel, 0.0));
#endif
#ifdef TF_PALETTE
    float voxel1 = texture2D(volumeSlice, pos.xy).r;
    float voxel2 = texture2D(volumeSliceN, pos.xy).r;
    float voxel = mix(voxel1, voxel2, pos.z);
    int index = int(voxel * 59.0);
    vec4 color = paletteLUT[index];
#endif
    return color;
}

void main()
{
    gl_FragColor = fetchVolume(vv3Texcoord.xyz);
}

```

## Ray casting shader programs

### *Vertex shader*

```

in vec4 av4position;
in vec4 av2texcoord;
in vec3 av3color;
out vec3 vv3color;
out vec4 vv2Texcoord;
out vec4 position;
uniform mat4 mvp;

void main() {
    vv2Texcoord = av2texcoord;
    vv3color = av3color;
    vec4 p = mvp * av4position;
    position = p;
    gl_Position = p;
}

```

### *Fragment shader*

```

precision highp float;
uniform highp sampler2D tflUT;
uniform vec4 paletteLUT[60];
uniform highp sampler2D backfaceTexture;
uniform highp sampler3D volumeTexture;
uniform highp sampler3D volumeTextureSmall;
uniform highp sampler3D volumeTexturePrecalc;
uniform highp sampler3D volumeTextureNormals;
uniform highp sampler3D volumeTextureSkipping;
uniform mat3 mv;
uniform vec3 viewerPos;
uniform vec3 lightPos;

in vec4 vv2Texcoord;
in vec3 vv3color;
in vec4 position;

```

```

out vec4 fragmentColor;
const vec3 specColor = vec3(1.0, 1.0, 1.0);

/* amount of steps for raymarching */
#define STEPS 512

vec4 fetchVolume(vec3 pos)
{
#ifdef TF_PRECALC_MODEL
    vec4 color = vec4(0.0, 0.0, 0.0, 0.0);
    if (distance(vec3(0.0, 0.0, 0.0), pos-vec3(0.5, 0.5, 0.5)) < 0.5)
        return vec4(1.0, 1.0, 1.0, 1.0);
#endif
#ifdef TF_PRECALC_SMALL
    vec4 color = texture(volumeTextureSmall, pos);
#endif
#ifdef TF_PRECALC
    vec4 color = texture(volumeTexturePrecalc, pos);
#endif
#ifdef TF_TEXTURE
    float voxel = texture(volumeTexture, pos).r;
    vec4 color = texture(tfLUT, vec2(voxel, 0.0));
#endif
#ifdef TF_PALETTE
    float voxel = texture(volumeTexture, pos).r;
    int index = int(voxel * 59.0);
    vec4 color = paletteLUT[index];
#endif
    return color;
}

vec3 calcNormal(vec3 pos)
{
    vec3 normal = vec3(0.0, 0.0, 0.0);
    vec2 diff = vec2(1.0 / 255.0, 0.0);
#ifdef TF_PRECALC_MODEL
    normal = pos - vec3(0.5, 0.5, 0.5);
#endif
#endif

```

```

#ifdef PRECALC_NORMALS
    // change 0..1.0 range to -1.0..1.0
    normal = (texture(volumeTextureNormals, pos).rgb - vec3(0.5, 0.5, 0.5)) * 2.0;
#endif

#ifdef FORWARD_NORMALS
    float s = texture(volumeTexture, pos).r;
    vec3 s1 = vec3(s, s, s);
    vec3 s2 = vec3(texture(volumeTexture, pos + diff.rgg).r,
                  texture(volumeTexture, pos + diff.grg).r,
                  texture(volumeTexture, pos + diff.ggr).r);
    normal = s2 - s1;
#endif

#ifdef CENTRAL_NORMALS
    vec3 s1 = vec3(texture(volumeTexture, pos - diff.rgg).r,
                  texture(volumeTexture, pos - diff.grg).r,
                  texture(volumeTexture, pos - diff.ggr).r);
    vec3 s2 = vec3(texture(volumeTexture, pos + diff.rgg).r,
                  texture(volumeTexture, pos + diff.grg).r,
                  texture(volumeTexture, pos + diff.ggr).r);
    normal = s2 - s1;
#endif

    return normal;
}

vec4 lightVoxel(vec3 N, vec3 L, vec3 V, vec4 color)
{
    float diffuse = max(dot(L, N), 0.0);
    float specular = 0.0;

    if (diffuse > 0.0)
    {
        // blinn phong
        vec3 H = normalize(L + V);
        float specAngle = max(dot(H, N), 0.0);
        specular = pow(specAngle, 16.0);
    }

    return vec4(diffuse * color.rgb + specular * specColor, color.a);
}

```

```

void main()
{
    vec2 screen=((position.xy / position.w) + 1.0) / 2.0;
    /* back coordinate from the rendered back faces */
    vec3 back = texture(backfaceTexture, screen).xyz;
    vec3 front = vv3color;
    /* calculate direction vector */
    vec3 dir = (back-front) / float(STEPS);
    vec3 pos = front;
    vec4 col = vec4(0.0, 0.0, 0.0, 0.0);
    /* ray casting loop */
    for (int i=0; i < STEPS; i++)
    {
        vec4 voxelColor=vec4(0.0, 0.0, 0.0, 0.0);
#ifdef SKIPPING
        /* read 3D volume skipping lut */
        vec4 skip = texture(volumeTextureSkipping, pos).rrrr;
        // check if we can skip or not
        if (skip.r > 0.008)
        {
#endif
            // read 3D volume
            voxelColor = fetchVolume(pos);

#ifdef LIGHT
            // do voxel local lighting
#ifdef LIGHT_SKIP_FULLY_TRANSPARENT
            if (voxelColor.a > 0.0)
            {
#endif
                // voxel normal
                vec3 N = -normalize(calcNormal(pos));
                // texture coordinate to model coordinate
                vec3 P = (pos - vec3(0.5, 0.5, 0.5));
                // vector from voxel to light
                vec3 L = normalize(lightPos - P);
                // vector from voxel to viewer
                vec3 V = normalize(viewerPos - P);
                voxelColor = lightVoxel(N, L, V, voxelColor);
            }
#ifdef LIGHT_SKIP_FULLY_TRANSPARENT
        }
#endif
    }
}

```

```
#ifdef LIGHT_SKIP_FULLY_TRANSPARENT
    }
#endif
#endif

    // front to back order blending
    col.rgb = col.rgb + (1.0 - col.a) * voxelColor.a * voxelColor.rgb;
    col.a = col.a + (1.0 - col.a) * voxelColor.a;
#ifdef EARLY_EXIT
    // early exit if color opaque enough
    if (col.a > 0.98)
    {
        // col.rgb = vec3(1.0, 0.0, 0.0);
        break;
    }
#endif
    pos+=dir;
#ifdef SKIPPING
    } else {
        i+=11; // 16 * sqrt(2) = 11
        pos+=dir * 11.0;
    }
#endif
    }
    fragmentColor = col;
}
```