Visualization of the Marked Cells of Model Organism

Radek Kubíček*

Department of Computer Graphics and Multimedia
Brno University of Technology
Brno / Czech Republic

Abstract

We will present the current state of ongoing work on a simple to use, quality result algorithm for visualization of marked cells of the model organism. In this work, we try to find a method based on volume rendering to visualize marked parts of input data composed by a set of confocal deconvolution microscope images in such a way that these marked parts can be higlighted and visualized. We have tried different techniques for dealing with the visualization speed and with the quality of the rendered images and propose best methods for realizing the work goals. In the last case, the quality of the rendered image is sufficient for imagination of position of the marked parts. We demonstrate results on different biological data sets, such as plant cells or model organism Caenorhabditis Elegans marked by GFP process.

Keywords: Volumetric Data Rendering, Biological Data Rendering, Shader, Pre-integration, Transfer Function

1 Introduction

Nowadays, direct volume rendering via 3D textures is an efficient tool for displaying and visual analysis of volumetric data sets. It is commonly accepted, especially for reasonably sized data sets, because of achieving interactive rates and appropriate visual quality. Unfortunately, there are also some problems, such as huge amount of per-fragment operations and need of big amount of memory. Real-time methods of direct rendering of volumetric data sets are still a challenge to the computer graphics community.

Volumetric rendering is acceptable and in most cases the one method for getting image of complexity and depth dependencies of volumetric data sets. This technique uses semi-transparent parts of the volume, which also makes it possible to view into the inner structures of the data. This technique is mostly used in biology and in medicine.

In biology, which is our case of use, volumetric images are used for the purpose of better understanding the inner structures within plant or animal body. The image diagnostic, new organism research, biology education and study of internal organism events are examples of applications that benefit from and require visualization of these images. Biology images are acquired mostly by special devices such as microscopes. Each image corresponds to a 2D image of a transversal cut of the organism, also called slice. The sequence of these images make a volume or a scene.

Suppose we have volumetric data set, captured by a confocal deconvolution microscope with some marked parts. Input data form one volumetric block containing separate slices. This data block we render by an applicable method and then we identify, highlight and visualize the cells marked by GFP (Green Fluorescent Protein) process [3, 8].

The principal aim of this work is to find of preferably optimally effective method enabling this highlight, mostly working without manual check. Due to the data structure, this ambition seems hard to be achieved, so it suffices to find a manually working method. This work also proposes different techniques for dealing with the rendering quality of images.

In the next sections, we describe the rendering techniques, usable in volumetric rendering. We also describe the selected re-slicing process and its comparison to other techniques. Finally, in Section 7, we examine the effects of these techniques and discuss the results and some performance issues.

2 Related Work

Real-time volume graphics has a long history in computer graphics, starting with the first straightforward CPU-based image-order approaches in 1984. With the increasing availability of hardware accelerated 3D textures, many volume rendering techniques have utilized this feature for different purposes. GPU-based image-order techniques have been developed only recently with the introduction of graphics boards that support true conditional branches and loops of variable length in the fragment processor. For it, there is existing GPU only based solution of direct rendering methods, mainly raycasting algorithm [12].

Since huge sizes of the volume data block are common in the volumetric rendering area, there have been also found some methods how to deal with large data sets that exceed the graphics adapter memory. In this work we

*xkubic23@stud.fit.vutbr.cz

probably will not use so large data sets, so we are not interested in these methods.

3 Used Techniques and Algorithms

Rendering techniques are used to transform 3D data into a 2D image, providing this resulting image with some 3D information such as linear perspective or shading. Several rendering techniques have been proposed that can be classified into two major classes: surface rendering and direct volume rendering. In surface rendering methods, only the voxels containing object boundary information are considered to contribute the rendering. It means that an approximate representation of the surface of the object is computed and rendered. In direct volume rendering methods, each voxel has assigned opacity value contributes to the rendering. The result is a semi-transparent volume. More about these techniques can be found in [1, 4].

In this paper we are concerned only about direct volume rendering methods, because they are used by the implementation and the surface methods are not probably suitable for this type of data, which is used in this work. At first, we have to define the way, how the data are saved in the memory. Next it is neccessary to select the rendering method to be used. In the next sections the techniques used in this state of the work can be found with their pros and cons. Input data, acquired by the confocal deconvolu-

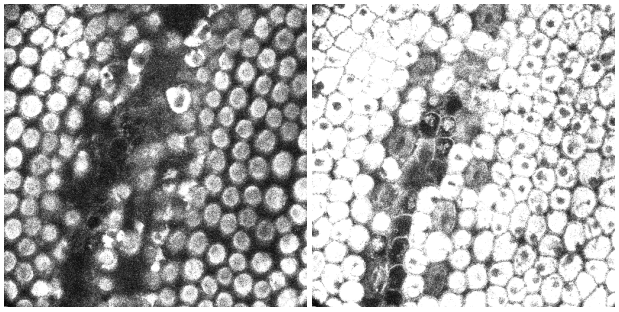

Figure 1: Strongly noised input data – plant cell structure

tion microscope are strongly noised, as illustrates Figure 1. It is necessary to use some de-noising filter, in the current state of the work we use the three-dimensional cross neighborhood median filter. The white areas in the image are the cells that we are visualizing. We need to use techniques that are capable with 16-bit data, for the input data are in this bit depth.

3.1 3D Texture Based Volume Rendering

In order for the graphics hardware to be able to access all the required volume information, the volume data must be downloaded and stored in textures. We use 3D texturing capability of graphics hardware, because in the rasterization step it is able to use trilinear filtering. Also 3D textures are well supported by all common graphics cards. One of limiting aspects of this approach is the fact that the amount of memory provided by most of commonly used graphics cards is still limited.

This approach stores the volume data in memory as one three-dimensional texture. With 3D textures the volume is usually split into viewport aligned slices. These slices are computed by intersecting the bounding box of the volume with a stack of planes parallel to the current viewport. In consequence, viewport-aligned slices must be recomputed whenever the camera position changes. Viewplane aligned polygons are drawn in back to front order or reversely. During rasterization, the transformed polygons are textured with the image data obtained from a solid texture block by trilinear interpolation. During fragment processing, the resulting polygon fragments are blended semi-transparently together into the frame buffer using alpha blending and finally displayed on screen [12]. Nowadays, there is no reason not to use 3D textures except for special cases needing 2D memory saved slices.

3.2 Re-slicing Method

Volume rendering via 3D textures is usually performed by slicing the texture block in back to front order with planes oriented parallel to the viewplane, see Figure 2. Slices are created in the following way. We make clip plane for each slice and then are calculated points of intersection with all 12 edges of volume data bounding box. Slice is created of 3 to 6 points. These points are sorted according to their angle and in this order they make a polygon. The coordinates of the points are also the texture coordinates. We map the volume texture on each polygon and then it is sent into the rasterizer and fragment shader. The resulting image quality depends strongly on the number of slices. More slices, better image quality. This algorithm has great

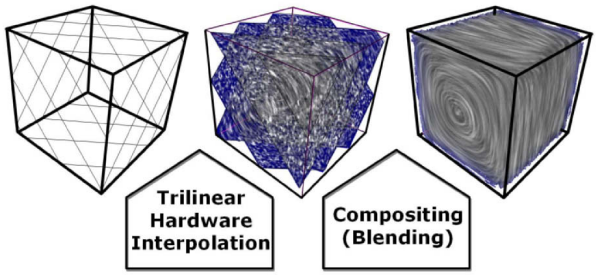

Figure 2: Volume rendering via 3D texture slicing [6]

advantage, it is very easy to implement it. It is only necessary to find a way how to get slices and how to generate the slices polygons. Simple vector analysis knowledge is sufficient. On the other side there is disadvantage in the fact, that small number of slices strongly degrades the quality of the resulting image due to fact, there is no available rasterized texture data for the whole volume space. It implies, that there is no interpolation and the resulting image seems

very rough. If we use only small number of the slices we could also miss the important details of the data. But if we use many slices, the rendering speed rapidly decreases, for the fragment shader is called very often and slows the result visualization.

If we compare implementation of the re-slicing method and ray-casting method, we find that slice-based volume rendering might be considered the "brute-force"-approach, that relies solely on the fill-rate and the high fragment throughput of the rasterization unit. Ray-casting on the other hand employs optimization techniques such as emptyspace skipping and early-ray termination. At the bottom line, however, the brute-force approach is still advantageous in terms of performance, while GPU-based ray-casting has several clear advantages when rendering iso-surfaces or sparse volumes [12].

We prefer the re-slicing algorithm for its easy of implemetation and for the high-quality results. There is one more advantage over the ray-casting method; the re-slicing algorithm uses the slices which consist of the vertices whereas ray-casting algorithm is generating a ray for every pixel of the scene. It means we can use the vertex shader for speed up of the re-slicing algorithm instead of the CPU-based slices generating method. The [12] demonstrates that the performance of object-order volume rendering can be improved by moving the necessary slice decomposition, which is usually done on the CPU, onto the vertex processor. This gives us the flexibility which is necessary to load-balance the rendering process for maximum performance. We will try to implement this improvement into the work.

4 Transfer Function

The role of the transfer function in direct volume rendering is essential. Its job is to assign optical properties to more abstract data values. It is these optical properties that we use for rendering a meaningful image. While the process of transforming data values into optical properties is simply implemented as a table lookup, specifying a good transfer function can be a very difficult task [1]. Why do we need a transfer function, i.e. why not store the optical properties in the volume directly? First, it is inefficient to update the entire volume and reload it each time the transfer function changes. It is much faster to load the smaller lookup table and let the hardware handle the transformation from data value to optical properties. Second, evaluating the transfer function at each sample prior to interpolation is referred to as pre-classification. Pre-classification can cause significant artifacts in the final rendering, especially when there is a sharp peak in the transfer function. An example of pre-classification can be seen on the left side of Figure 3. A similar rendering using post-classification is seen on the right [1].

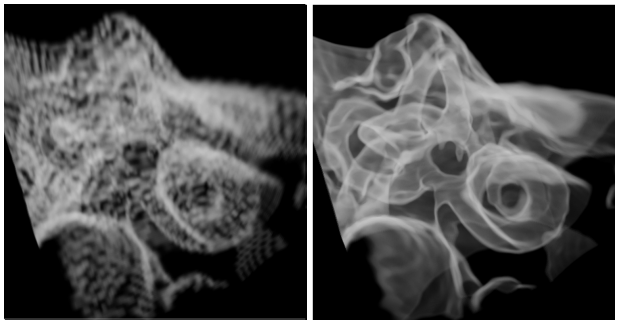

Figure 3: Pre-classification verses post-classification [1]

Transfer functions are usually represented by color lookup tables. They can be one-dimensional or multi-dimensional, and are usually stored as simple arrays. Transfer functions may be downloaded to the hardware in basically one of two formats: In case of the pre-classification, transfer functions are downloaded as texture palettes for on-the-fly expansion of palette indides to RGBA colors. If the post-classification is used, transfer functions are downloaded as 1D, 2D, or even 3D textures (the latter two for multidimensional transfer functions). If pre-integration is used, the transfer function is only used to calculate a pre-integration table, but not downloaded to the hardware itself. Then, this pre-integration table is downloaded instead [1]. By using the transfer function the users can set the optical properties of the volume. Figure 4 shows the tranfer function editor used in this work. On bottom there are buttons for switching the active channels. By moving mouse through the graph area we can specify values for the active channels. Then these values are used in the pre-integration process for calculating the pre-integration dependent texture. In the graph area there is the volume data histogram also visible which can be used for the best transfer function estimation.

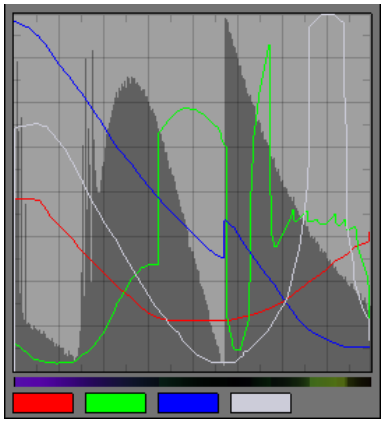

Figure 4: Transfer function editor

The same transfer function principle is also used for adjusting the axis slices contribution function but instead of calculating the resulting function of all the values it is used to save each curve into the relevant axis contribution one-dimensional texture.

One of the possible enhancements of the transfer function could be use of the logarithmic scale. For the fact that the bands span an extremely narrow region for depths above 30 percent of the volume means that for such regions, differences in transfer function values as small as 0.001 (in the region near zero) will visibly change the image. Even if an entire screen is used for a transfer function editor, the user must be able to specify the transfer functions to the accuracy of a single pixel to make such changes. At the same time, values above 0.05 all map to nearly opaque, resulting in 95 percent of the screen space being wasted. If we scale the transfer function opacities logarithmically, it would be easier, both conceptually and physically, for a user to precisely control the intensity of a region in the volume. This should result in more efficient transfer function editing, which is significant since transfer function editing is one of the most time-consuming aspects of creating clear and informative renderings of medical, mathematical or generally scientific data sets [9]. More about the transfer function principle and directions how to create the extended or the multi-dimensional transfer functions is described e.g. in [1].

5 Pre-integration

High accuracy in direct volume rendering is usually achieved by very high sampling rates, because the discrete approximation of the volume rendering integral will converge to the correct result for a small slice-to-slice distance $d \to 0$, i.e., for high sampling rates $n/D = 1/d$. However, high sampling rates result in heavy performance losses, i.e. as rasterization requirements of the graphics hardware increase, the frame rates drop respectively. According to the sampling theorem, a correct reconstruction is only possible with sampling rates larger than the Nyquist frequency [1]. It means, if the two near-by slices are too far from each other, there can be a little but important detail in the data and we do not get it in the resulting image. If the sampling distance is not at least as large as minimal important detail in the data, we can miss this detail during the sampling.

The pre-integration provides high image quality post-classification method even with low-resolution volume data. Besides direct volume rendering, the algorithm also allows us to render double-sided isosurfaces with diffuse and specular lighting without extracting a polygonal representation. More about this technique can be found in [2, 1, 10]. The main principle of the pre-integration algorithm is illustrated on Figure 5. Let us have the slice that is actually rendered. Imagine we cast a ray through each pixel of this slice. Sampling distance is the distance, this ray takes until it reaches the next slice. But we want to know optical values along the whole ray. It is allowed by the pre-integration function. We can create slabs, imaginary spaces between near-by slices. As we can see on Figure 5, the transfer function determines the optical values

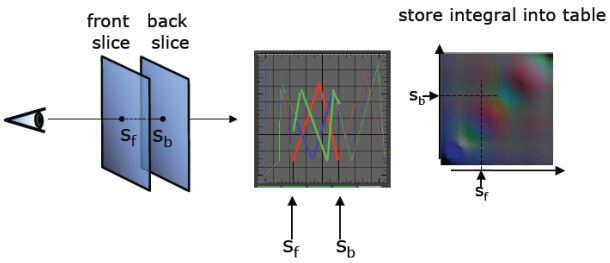

Figure 5: Pre-integration algorithm principle [7]

for all possible data values. During the pre-integration process, we calculate all possible combinations in the transfer function for front and back slices and these values are then saved into integral transfer table. This table makes a new 2D texture, that is sent into the graphics card texture memory. During the fragment operation, we fetch values for the current and previous slice (if we render in back-to-front order) and using them we determine the final color and opacity of the rendered pixel.

The primary drawback of pre-integrated classification in general is actually the preintegration required to compute the lookup tables, which map the three integration parameters (scalar value at the front, scalar value at the back, and length of the segment) to pre-integrated colors and opacities. As these tables depend on the transfer functions, any modification of the transfer functions requires an update of the lookup tables [1]. This causes that the pre-integration texture could be updated either after the transfer function modification or during the modification, but in this case there are executed many costly computations and the modification is very difficult to be performed interactively. Also it can be helpful to implement an accelerated pre-integration algorithm. This approach has not so quality result as full pre-integration, but it is much faster and allows to imagine how the result will look. After the modification of the transfer function the full pre-integration algorithm is then executed. Some of these methods are described in [1, 10].

In this work we use this technique for setting the optical properties as color and opacity with the helpful transfer function. This transfer function we pre-integrate into 2D texture and then this texture is used in fragment shader for getting demanded optical properties. Probably we could use only opacity channel if the color palette is specified by the application. Pre-integration is still not adaptable to the change of the number of slices – the sampling distance. It is necessary due to increase or decrease of the amount of the optical properties, because if the number of slices is huge, more values are integrated and maximal value is reached earlier. The next problem is, that the data is 16-bit but we use the transfer function with only 256 entries (8-bit range), because it is rendered onto viewplane with the 256 pixels size. So we need to find a way, how to adapt the transfer function to the 16-bit range. We can use the current transfer function editor, but, due to using 8-bit range

we will have the same value for 256 possibly different values of 16-bit range image. It seems that zoomable transfer function editor could be the right approach. It means we will have 16-bit range transfer function for 16-bit volume data and for its size is huge for render on screen it needs to enable zooming and set detail values in the transfer function for the right zoom level. Probably it will be also helpful to use iso surfaces and the iso surface pre-integration, described i.e. in [10, 11].

6 Fragment Shader

After creation of the texture mapped slices, they are sent into the rasterization. Of course, we can use rendering pipeline with fixed functionality, but we do not get the increased functionality and the most of all the GPU hardware accelerated rasterization. The fragment shader enables the accelerated rendering. We use the Cg language and runtime environment to do this job, but if another language or approach will prove to be more convenient, we can use it as well. In Figure 6 we can see the core of the

```
 1 uniform float       uIsoValue;
 2 uniform sampler3D   uTexVolume;
 3 uniform sampler2D   uTexTransfer;
 4 uniform sampler1D   uTexSlX;
 5 uniform sampler1D   uTexSlY;
 6 uniform sampler1D   uTexSlZ;
 7
 8 void fshader_main(
 9   float3 iTexCrd0   : TEXCOORD0,
10   float3 iTexCrd1   : TEXCOORD1,
11   float4 iColor     : COLOR,
12   out float4 oColor : COLOR)
13 {
14   float4 slX = tex1D(uTexSlX, iTexCrd0.x);
15   float4 slY = tex1D(uTexSlY, iTexCrd0.y);
16   float4 slZ = tex1D(uTexSlZ, iTexCrd0.z);
17   float3 slices = float3(slX.x, slY.x, slZ.x);
18
19   if (all(slices))
20   {
21     float val1 = tex3D(uTexVolume, iTexCrd0).r;
22     float val2 = tex3D(uTexVolume, iTexCrd1).r;
23
24     float4 color = tex2D(uTexTransfer,
25                          float2(val1.r, val2.r));
26
27     float multCoeff = slices.x*slices.y*slices.z;
28     oColor = color*multCoeff;
29   }
30   else
31   {
32     discard;
33   }
34 }
```

Figure 6: Fragment shader inner algorithm core

fragment shader algorithm. This algorithm core is as much as possible flexible piece of code, so it can be enhanced by the more efficient and complex code in the future. Because the fragment shader makes a lot of work, it is the most critical part of the renderer. It is also possible to optimize and speed up this algorithm, as well as add more functionality, due to the fact, that the fragment shader is executed in parallel and it is strongly hardware accelerated. Also, nowadays most of the graphics cards are capable of using fragment shaders.

Lines $1-6$ are declarations of the global (also called uniform) parameters. Property *uIsoValue* is not used yet, but it is implemented here for possible future use of the iso surfaces. *uTexSlX*, *uTexSlY* and *uTexSlZ* determine the contribution of the slices in each axis of the volume and they are also adjustable by the user. The test, if the currently rendered pixel has nonzero contribution on each axis is on the line 19. Parameters *uTexVolume* and $uTexTransfer$ are pointers to the used volume and transfer textures in the memory. These contributions of the currently rendered pixel are fetched for each volume axis on lines $14-17$. If the currently rendered pixel has any contribution to the resulting image, values on the lines $21-22$ are fetched for this pixel for the rendered slice and the previous slice. Then, on lines $24-25$, these values create the texture coordinates into transfer textures and the optical properties are fetched from the pre-integrated transfer texture. On line 27 the total pixel contribution coefficient is calculated and at the end, on line 28 we set the optical values multiplied by this coefficient as the resulting value for this pixel. If the current pixel has zero contribution to the final image, so the test on lines $19-20$ failed, we discard the current pixel on line 32. Then this pixel does not affect the resulting image and the texture values are not fetched, so it slightly increase the speed of the rendering. The current implementation uses *discard* statement; it could also use the $clip()$ operation, or in lower shader profiles set the pixel to black, fully transparent value for this purpose.

7 Results and Future Work

However the current status of this work is still experimental, we are able to visualize medical or synthetical data. The volume renderer has been created operating the most efficiently on 8-bit data. 16-bit data can be also visualized, but the application is still not optimized for this size of the data elements. This volume renderer works fine especially for the synthetic data, but it is only necessary to implement some described improvements and it will work fine for the biological and medical data. The hardware acceleration algorithm core is finished and it remains to implement some enhanced functionality or modify this core according to our demands. It can be also easy to implement additional shading or illumination algorithm if necessary. It could be done using the fragment shader capabilities of the reading texture values. Fragment shader could estimate the gradient vector on-the-fly using e.g. central differences and multiple texture lookups per fragment (8 texture lookups per fragment). It also remains to implement some function for the marked parts of the input data highlighting, which should be the main part of this work.

In the current state of the work, after rendering the input biological data, the result looks like on Figure 7 and 8. As

we can see, there are partly visible marked cells of data, but they are not so much differentiated from the other data, as they should be. If the border slices which contain most of all noise are not visualized (Figure 7), marked cells are visible more than if these border slices are rendered too (Figure 8). For comparison we can see on Figure 9, that if we render non-noised data, the quality of the resulting image is very good.

Despite the noise in the input data, which must be pre-filtered by some noise removing filter, visual quality of result images is quite good and later it should be comparable to the original data images. We need to propose an algorithm for quality removal of noise and also for differentiating marked parts of data. We probably also need to use some of the global filtering or classifying function before the visualization. The speed of the algorithm enables a real-time visualization of the slices, but only if the number of slices is not very huge and sizes of data are adequate. It is also necessary to optimize the pre-integration function according to the number of slices.

Implementation of a function of each slice contribution to the result image also seems to be very helpful. Because when we have a detail look into data structure we get the fact, border slices are roughly noised and very degradate the result image. This function is a one-dimensional curve for each axis. This curve sets contribution ratio for each slice from the $[0.0, 1.0]$ range. Then this function is saved into one-dimensional texture which is sent into the graphics card memory and used by the fragment shader. These values of contributions are adjustable by the user using the transfer function. It could be also considered as another way how to deal with the global image statistical functions.

8 Conclusions

In this paper, we have briefly presented the current state of a technique for visualization of marked cells in volumetric data sets based upon the direct volume rendering algorithms. It consists of a re-slicing method and GPU-based acceleration by fragment shaders. We also use the pre-integration algorithm to set the optical properties for each value of the volume. As next, we have suggest some methods, how to deal with the noised data and poor quality slices that degrade the resulting image. Also we have outlined some principles and methods how speed up this work and enhance the results.

References

[1] Klaus Engel, Markus Hadwiger, Joe M. Kniss, and Christof Rezk-Salama. *High-Quality Volume Graphics on Comsumer PC Hardware*. SigGraph Course Notes 42, 2002.

[2] Klaus Engel, Martin Kraus, and Thomas Ertl. *High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading*. Visualization and Interactive Systems Group, University of Stuttgart, Germany, 2001.

[3] Jay Enten and Brendan Yee. *Green Fluorescent Protein (GFP)*. Beckman Coulter, Inc., Miami, FL, 2005.

[4] Sören Grimm. *Real-Time Mono- and Multi-Volume Rendering of Large Medical Datasets on Standard PC Hardware*. Technischen Universität Wien, Fakultät für Informatik, 2005.

[5] Evan Hart. *3D Textures and Pixel Shaders*. ATI Research, 2004.

[6] J. Krüger and R. Westermann. *Acceleration Techniques for GPU-based Volume Rendering*. Computer Graphics and Visualization Group, Technical University Munich, 2003.

[7] Kwan-Liu Ma. *An Efficient Pre-Integrated Volume Rendering Algorithm*. Department of Computer Science, University of California at Davis, 2006.

[8] Colm O'Carroll. *Green Fluorescent Protein*. B/MB senior seminar, 2006.

[9] Simeon Potts and Torsten Möller. *Transfer Functions on a Logarithmic Scale for Volume Rendering*. Graphics, Usability and Visualization (GrUVi) Lab, School of Computing Science, Simon Fraser University, 2002.

[10] Stefan Röttger and Thomas Ertl. *A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids*. Visualization and Interactive Systems Group, University of Stuttgart, Germany, 2002.

[11] Stefan Röttger, Martin Kraus, and Thomas Ertl. *Hardware-Accelerated Volume And Isosurface Rendering Based On Cell-Projection*. Visualization and Interactive Systems Group, University of Stuttgart, Germany, 2002.

[12] Christof Rezk Salama and Andreas Kolb. *A Vertex Program for Efficient Box-Plane Intersection*. Computer Graphics and Multimedia Systems Group University of Siegen, Germany, 2005.

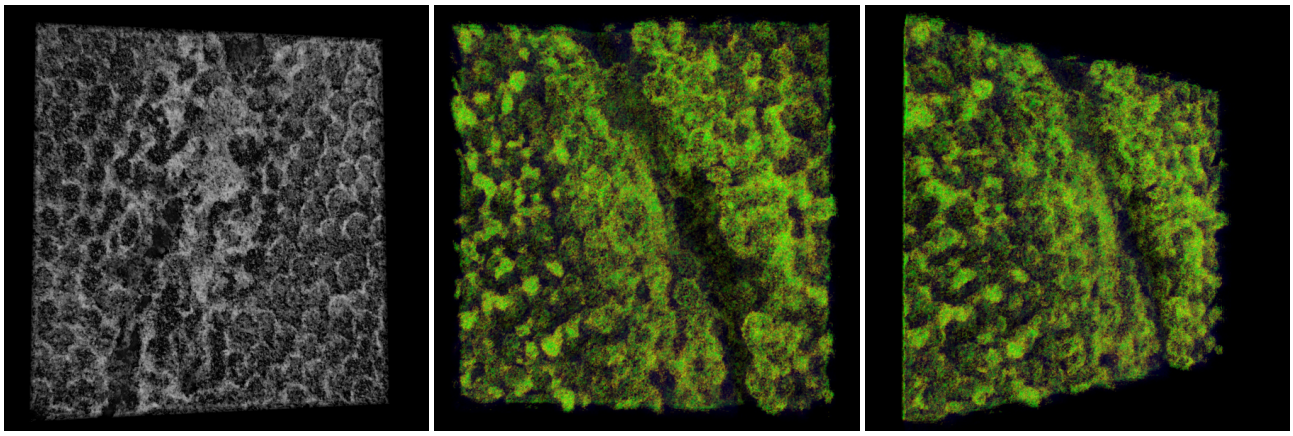

Figure 7: Resulting images - plant cell structure. Image size is 512x512x92x16b. Not all slices rendered.

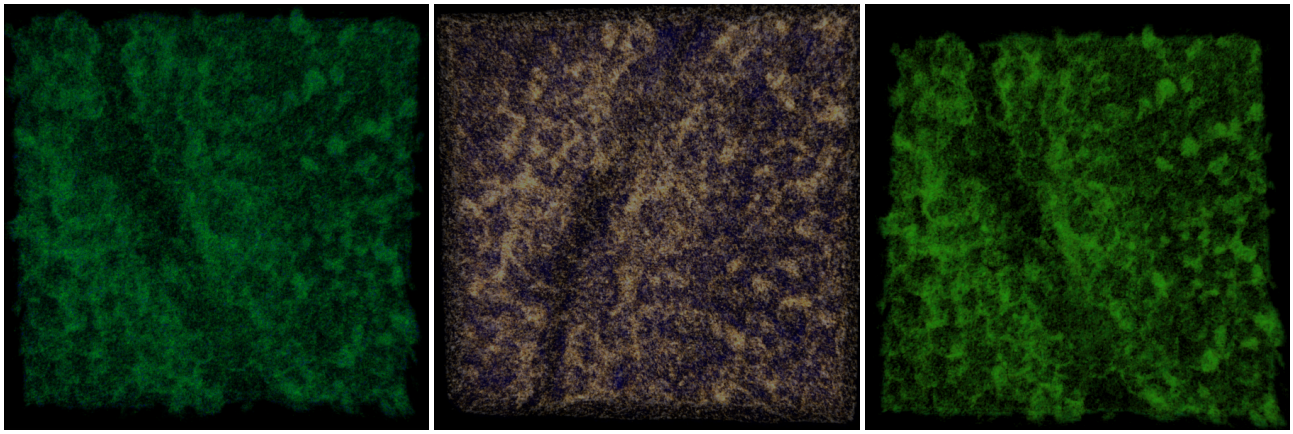

Figure 8: Resulting images - plant cell structure. Image size is 512x512x92x16b. All slices rendered.

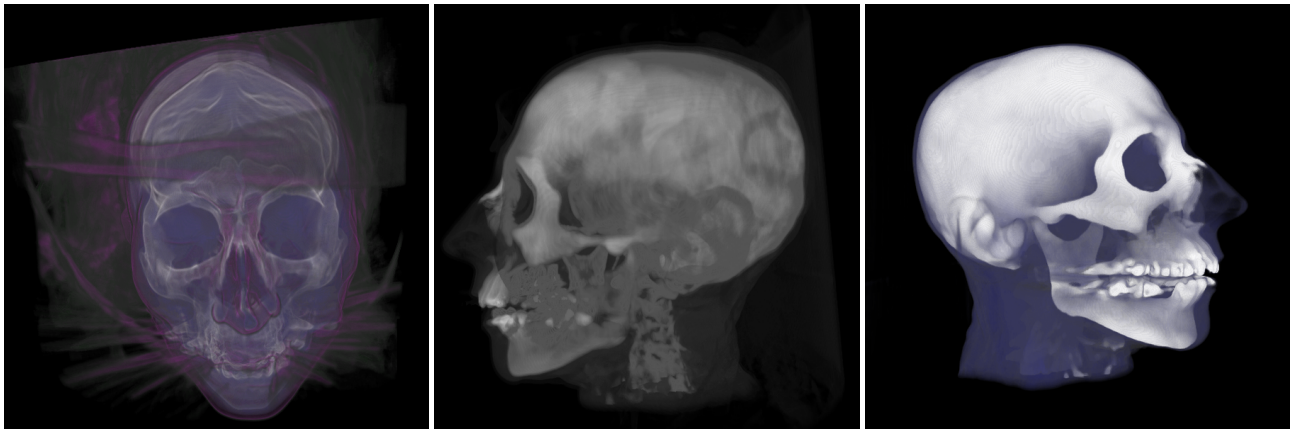

Figure 9: Resulting images - CT head data. Image size is 256x256x225x8b. All slices rendered.