Scientific Visualization, 2019, volume 11, number 5, pages 1 - 11, DOI: 10.26583/sv.11.5.01

Memory-effective methods and algorithms of shader visualization of digital core material model

M.V. Mikhaylyuk[1], P.Yu. Timokhin[2]

Federal State Institution "Scientific Research Institute for System Analysis of the Russian Academy of Sciences" (SRISA)

[1] ORCID: 0000-0002-7793-080X, mix@niisi.ras.ru
[2] ORCID: 0000-0002-0718-1436, webpismo@yahoo.de

Abstract

The paper deals with the task of real-time visualization of digital core material model (DCM) obtained by X-ray computed tomography. Novel approach, effective methods and algorithms for visualization of large allocated volume of DCM are proposed, which are based on high-performance programmable tessellation capability of commodity graphics cards with a multicore graphics processing unit (GPU). The methods provide an effective video memory usage and a high visualization rate for the ranges of absorption coefficient, corresponding to pore space or mineral skeleton of the core material. The solution proposed is based on visualization of trihedral polygonal models of visible voxels and on the original system for constructing such models on GPU by means of tessellation shaders developed. Based on these methods and algorithms, a program module is developed, which implements 3D visualization of the allocated volume of DCM and the construction of porosity plot. The program module was tested on the DCMs of Bazhenov formation and sandstone, that confirmed adequateness of developed solution to the task arisen.

Keywords: visualization, digital core model, real-time, voxel rendering, pore space, porosity, GPU, tessellation, shader.

1. Introduction

One of the important tasks in the field of up-to-date oil and gas engineering is computer research of pore space and mineral skeleton of the core material [1]. As the basis of the research a digital core material model (DCM) obtained by X-ray computed tomography is used. Microtomograph determines the values of X-ray absorption coefficient for the core material in 3D grid cells, after that these values are scaled and saved in special file (file data set).
Effective means to extract scientific knowledge from such data is visual analysis which point is to present large amounts of data in a form where a researcher could see things that are difficult to identify algorithmically. Presently, the development of effective methods of the visual analysis of large volumes of digital data is being carried out by many reputable research teams, in particular, one can highlight an interesting method of elastic maps for the visual analysis of multidimensional data sets of various origins [2]. In the field of visual analysis of digital core material models, various methods for 3D visualization of virtual models of pore space and mineral skeleton are being actively evolved [3]. One of the basic ones is the visualization of the allocated volume in a virtual scene, which comprises only the cells of DCM, that correspond to the range of absorption coefficient values, specified by a researcher. These cells are visualized as cubes (voxels), where each cube is rendered taking into account its illumination in the virtual scene, and the color of the cube corresponds to absorption coefficient value in this cell.

To carry out the visual analysis of DCM effectively, the visualization of allocated volume should be real-time (with a frame rate of at least 25 times per second), as well as the volume should cover an area of DCM large enough to be representative. The challenge is, that rendering time and video memory consumption highly increase, when size of allocated volume is getting bigger. Therefore, visualization rate critically decreases for wide ranges of absorption coefficient corresponding to pore space or mineral skeleton of the core material. For this reason, a task of development of new approaches, methods, and algorithms allowing to overcome these restrictions, and basing on modern graphics hardware and software capabilities is arisen.
In this paper we propose a novel approach and effective methods, and algorithms, which implement real-time visualization of large allocated volumes of DCM on a personal computer equipped with a commodity graphics card with a multi-core graphics processor (GPU). The solution proposed is based on visualization of trihedral polygonal models of visible voxels which are created on GPU in parallel and independently from each other, by means of original shader computing system.

2. Approaches to visualize a digital core material model

One of conventional approaches to visualize voxel data, in particular, a digital core material model, is "ray casting" [4, 5]. A ray is being emitted through every pixel of synthesized image of allocated volume (generally it is a parallelepiped) until this ray crosses the first cell satisfying sampling condition (the cell's absorption coefficient value falls within the range specified by a researcher). Searching for such intersection is based on checking the cells which the ray traverses. This means that visualization time of a single voxel highly depends on the size of allocated volume, and the number of affected pixels, as well as the screen resolution. These dependencies are time-consuming for real-time rendering of large allocated volume (of 1000^3 cells and higher) on modern screens with high resolution (Full HD, Ultra HD). Due to this, different accelerating techniques and data structures are used, that needs additional memory and extra-time for data preprocessing. Consumption of these resources rapidly increases with the rising of volume size, that significantly limits application of the approach on personal computers (about 16 GB RAM, 11GB VRAM). For instance, well-known program complex Paraview [6] spends about 5Gb RAM to render an allocated volume of 600^3 cells with a small sampling range of absorption coefficient (about 800 counts). And if we specify a wider sampling range, for example, for pore space (about 9000 counts), then memory consumption will come out of the limits.
Another approach is to visualize the cells of sampling by means of polygonal (triangulated) cubic models [7-9]. Its efficiency depends on consumption of time and video memory for construction and visualization of a single polygonal cube. In the age of multicore GPUs using a hierarchical VRAM structure, the bottleneck of visualization of multiple simple polygonal models is the number of invocations to global video memory (the largest and the slowest part of VRAM). One way [7] to reduce it is to store in global video memory all polygonal cubic models (positions of vertices, colors and normals), to be rendered, as continuous arrays - vertex buffer objects (VBO), which are read by GPU very efficiently. However, this way is accompanied by extensive video memory costs (about 300 bytes per voxel), that limits its applicability to visualization of only small and medium volumes. Effective means to evade this limit is to use a geometry shader - one of the programmable stages of GPU graphics pipeline. This allows to construct a simple polygonal model for every voxel, in parallel, directly inside the graphics pipeline (without storing in global video memory), for instance, like in work [8]. To start one geometry shader thread, one vertex is enough to be sent to pipeline, therefore the VBO could be decreased up to 1 vertex per cubic model. Although, video memory consumption is greatly reduced, the number of invocations of vertex shader (the first stage of the graphics pipeline, which addresses directly global memory) is still too high to visualize large allocated volumes in real-time.

Relatively recently this was a serious limitation, till the two new programmable stages - the tessellation control shader and the tessellation evaluation shader were added to the graphics pipeline [10]. These stages are executed before geometry shader and are intended, in particular, to create, in parallel, regular grids of conventional graphics primitives out of special parametric square primitives (patches). Based on tessellation shaders capabilities, new methods and algorithms to visualize large allocated volume of DCM were developed in this paper, so it allows to reduce VBO size and the number of vertex shader invocations up to 4 vertices (invocations) per 65^2 cubic models. The Section 3 of the paper describes the developed method for construction of an effective polygonal model of visible cube surface, called "trefoil", which we create in geometry shader. The Section 4 deals with developed methods and algorithms of parallel generating of vertices by means of tessellation shaders for starting geometry shader threads.

3. A method to construct visible polygonal cube surface

Let us consider the task of visualization of some volume M extracted from 3D array of absorption coefficient values of DCM. The researcher specifies the volume in graphical user interface by setting coordinate segments $\left[x_{begin}, x_{end}\right] \times \left[y_{begin}, y_{end}\right] \times \left[z_{begin}, z_{end}\right]$. Let's introduce the following designations $n = x_{end} - x_{begin}$, $m = y_{end} - y_{begin}$, $q = z_{end} - z_{begin}$ and denote by K_a the 3D array of absorption coefficient values in voxels of volume M. Inside the volume M we will visualize only that voxels which absorption coefficient $k_a \in \left[k_{a,\min}, k_{a,\max}\right]$ (the range is also specified by the researcher).

Denote by P_{eye} the observer position in the virtual scene. Consider an arbitrary voxel (cube) of unit size from the volume M. Enumerate cube's vertices as follows: 0 – $(0,0,0)$, 1 – $(1,0,0)$, 2 – $(0,1,0)$, 3 – $(1,1,0)$, 4 – $(0,0,1)$, 5 – $(1,0,1)$, 6 – $(0,1,1)$, 7 – $(1,1,1)$. No matter, from which angle the cube will be observed, one will see no more than three faces of the cube. Actually, if voxel is located lower and to the left of P_{eye} point, then observer will see right, top and front faces of the cube. Such three faces are unambiguously defined by their one common vertex and are formed in a figure, which we will call as a "trefoil". Note, that if observer sees two (or even one) faces, then we add to observing one (two) degenerated faces. It is easy to see, that there are 8 various variants of a "trefoil" (see Fig. 1.).

To determine which variant of a "trefoil" is seen from P_{eye} position, we will check in View Coordinate System (VCS, [11]) the cosines of the angles between $\vec{e}_{vcs} = (0,0,1)$ (inversed viewing direction of the observer) and normals $\vec{n}_{vcs,0}$, $\vec{n}_{vcs,1}$, $\vec{n}_{vcs,2}$ to the right, top and front face of the cube. These normals are the same for all voxels for current visualization frame, and may be calculated as $\vec{n}_{vcs,p} = M_{norm}\vec{n}_{ocs,p}$, where M_{norm} is normal matrix, computed by taking the transpose of the inverse of the upper-left 3×3 submatrix of the ModelView matrix [11], and $\vec{n}_{ocs,0} = (1,0,0)$, $\vec{n}_{ocs,1} = (0,1,0)$, $\vec{n}_{ocs,2} = (0,0,1)$ are normals to the right, top and front face of the cube in Object Coordinate System (OCS). It is easy to notice, that $\vec{n}_{vcs,p}$ will coincide with M_{norm} matrix columns. Note, that normals $\vec{n}_{vcs,0}$, $\vec{n}_{vcs,1}$ and $\vec{n}_{vcs,2}$ should be normalized after calculation.

The cosine of the angle between $\vec{e}_{vcs}$ and $\vec{n}_{vcs,p}$ will equal to $\left(\vec{e}_{vcs}, \vec{n}_{vcs,p}\right) = \vec{n}_{vcs,p,z}$. Then number t of visible "trefoil" may be calculated as

$$t = b_0 + 2b_1 + 4b_2, \tag{1}$$

where $b_p = 1$, if $\vec{n}_{vcs,p,z} \geq 0$, and $b_p = 0$ otherwise ($p \in \{0,1,2\}$).

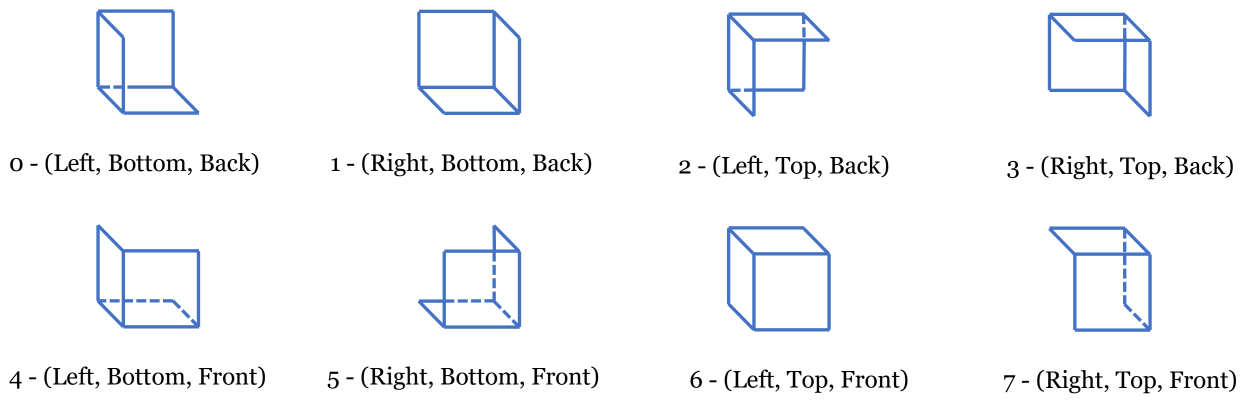

Fig. 1. Possible variants of a “trefoil”.

Every “trefoil” S we will visualize as a triangle strip. The strip is written as vertex number sequence where every three following vertices form a triangle. For our “trefoils” we get the next strips: $S_0=(5,4,1,0,3,2,2,0,6,4)$, $S_1=(4,0,5,1,7,3,3,1,2,0)$, $S_2=(1,0,3,2,7,6,6,2,4,0)$, $S_3=(5,1,7,3,6,2,2,3,0,1)$, $S_4=(1,5,0,4,2,6,6,4,7,5)$, $S_5=(0,1,4,5,6,7,7,5,3,1)$, $S_6=(0,4,2,6,3,7,7,6,5,4)$, $S_7=(4,5,6,7,2,3,3,7,1,5)$. Each of these strips contains two degenerated triangles which are not visualized and don’t consume computing resources.

In order to visualize the “trefoil” properly (with correct illumination of its faces), it is necessary to set the same normal (corresponding to cube face) for all vertices of every triangle in strip. This is not a trivial task, due to the fact that every next triangle in the strip uses two vertices from the previous one. To resolve this challenge, we use a technique called “Provoking Vertex” [11] supported by all modern graphics cards. Its point is, if one disables interpolation mode for some vertex attribute, for instance, a color (by setting the shading mode to “flat-shading”), then the entire graphics primitive will have the color of the last vertex (by default) called “provoking vertex”. In triangle strip of “trefoil”, the “provoking” is every $(g+2)th$ vertex, where $g=1,2,\ldots,8$ is running number of a triangle in the strip.

In this paper, we visualize the strips of “trefoils” with disabled interpolation of vertex normals. Prior to this we specify the normals to cube faces for the “provoking” vertices of the strip. For this, we enumerate the normals to the faces as follows: 0 is normal $\vec{n}_{vcs,0}$ to the right face, 1 – normal $\vec{n}_{vcs,1}$ to the top face, 2 – normal $\vec{n}_{vcs,2}$ to the front face, 3 – normal $-\vec{n}_{vcs,0}$ to the left face, 4 – normal $-\vec{n}_{vcs,1}$ to the bottom face, 5 – normal $-\vec{n}_{vcs,2}$ to the back face. For triangle strips $S_0,\ldots,S_7$, we obtain the following sequences of normal's numbers: $N_0=(*,*,4,4,0,0,0,0,5,5)$, $N_1=(*,*,5,5,1,1,1,1,3,3)$, $N_2=(*,*,4,4,2,2,2,2,0,0)$, $N_3=(*,*,3,3,1,1,1,1,2,2)$, $N_4=(*,*,4,4,5,5,5,5,3,3)$, $N_5=(*,*,0,0,1,1,1,1,5,5)$, $N_6=(*,*,4,4,3,3,3,3,2,2)$, $N_7=(*,*,2,2,1,1,1,1,0,0)$, where “*” denotes an arbitrary number of the normal (when processing the “provoking” vertices of the strip, the first two are skipped).

To visualize the allocated volume M, we will create GPU threads (one for each voxel), which we will call “voxel threads”, and the sequence of shaders working in each stream. Consider it in detail.

4. A method to calculate voxel threads using shader tessellation

Each “trefoil” will be visualized in a separate GPU thread. To create the threads, it is proposed to use programmable tessellation (subdivision into triangles) of graphics primitives. As a primitive, to be tessellated, we take a square (*patch-quad*) and divide it into l^2 small sub squares, which are formed by newly created $(l+1)^2$ vertices (see Fig. 2). In up-to-date graphics cards the maximum value of l is guaranteed to be at least 64, so we use it in our tests.

Further, for every vertex a GPU thread will be created. The threads obtained from a single patch-quad will form a *group of voxel threads*. Groups can be represented as one-dimensional array of size $n_P \times m_P \times q_P$, where $n_P = \lceil n/(l+1) \rceil$, $m_P = \lceil m/(l+1) \rceil$ and $q_P = q$. Thus, the number of patch-quads we need will also be equal $n_P \times m_P \times q_P$. In addition, our solution uses the following texture-based structures:

- **Bit map of voxels** – 3D bit array $B[i, j, k]$ of $n \times m \times q$ size, where bit value equals 1, if absorption coefficient value k_a in the corresponding voxel belongs to the sampling range $[k_{a,\min}, k_{a,\max}]$.
- **Bit map of patch-quads** – 3D bit array $B_P[i_P, j_P, k_P]$ of size $n_P \times m_P \times q_P$, where bit value equals 1, if there is at least one voxel in the corresponding group of voxel threads, satisfying the sampling condition.
- **Color palette** – array A of 256 colors, where every absorption coefficient value is corresponded to the specified color of palette.
- **Color index map** – 3D array $I[i, j, k]$ of size $n \times m \times q$, where $I_{i,j,k}$ represents indexed color of $(i, j, k)th$ voxel (we use 1-byte color indices).

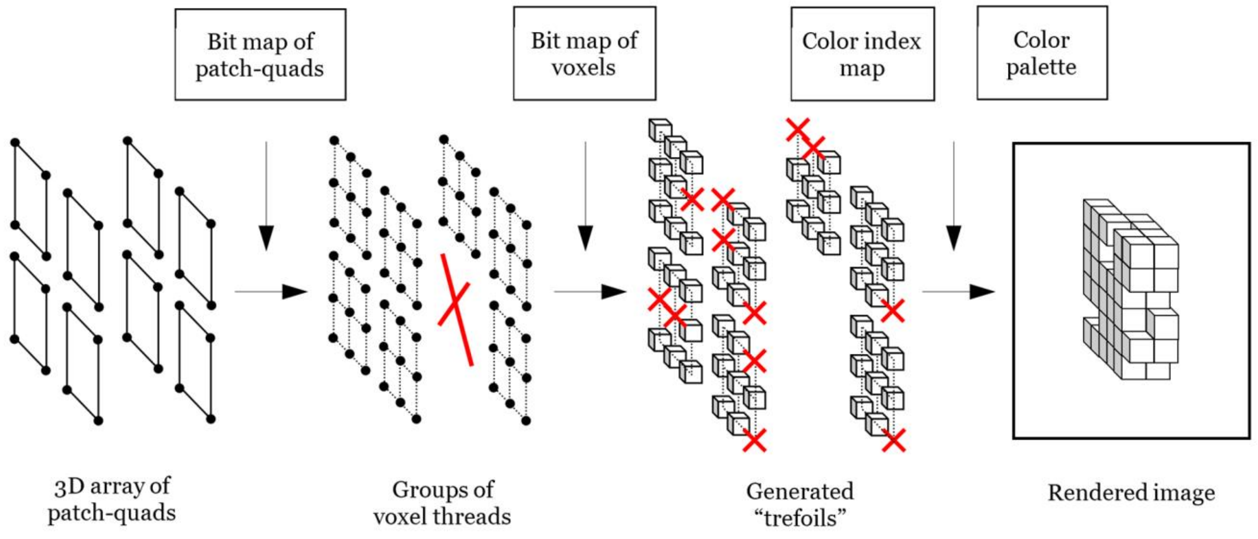

Fig. 2. Our visualization workflow.

The array of patch-quads will be sent to the graphics pipeline in each visualization frame and processed by means of developed complex of shader programs. The complex includes tessellation control (TC) shader, tessellation evaluation (TE) shader, geometry (G) and fragment shader.

TC-shader runs simultaneously on all GPU cores and processes its own patch on each core. TC-shader computes three indices (i_P, j_P, k_P) of the patch (as if the patch was stored in a 3D-array of size $n_P \times m_P \times q_P$), as well as the numbers l_W and l_H of sub squares into which the patch will be subdivided during the tessellation. After the TC-shader is executed, the primitive generator subdivides the patch (called further as original patch) into a group of vertices (the initiators of voxel threads) of size $(l_W+1)\times(l_H+1)$. This is a fixed stage of the graphics pipeline with a hardware-implemented algorithm. Generated vertices are given by positions (u,v) in the original patch, where u,v are normalized real coordinates, $u,v\in[0,1]$. After subdividing the original patch into vertices, TE-shader is performed which for each vertex of the vertex (thread) group generated calculates three voxel indices (i,j,k) in the 3D-voxel array. The whole process described is implemented in the following

Algorithm of generation of voxel threads

1. Transfer to TC-shader the following parameters:

- n_P, m_P, q_P - the dimensions of 3D-array of patch-quads;
- $n_{slice} = n_P m_P$ - the number of 2D-arrays of patch-quads (slices) in 3D-array of patch-quads;
- $n_{patches} = n_P m_P q_P$ - the size of 3D-array of patch-quads;
- l - the maximum number of sub quads along a single side of the patch-quad.

2. Calculate indices (i_P, j_P, k_P) of the patch:

$$k_P = \left\lfloor \frac{id}{n_{slice}} \right\rfloor, \quad i_P = \left\lfloor \frac{id - k_P n_{slice}}{n_{patches}} \right\rfloor, \quad j_P = id - k_P n_{slice} - i_P n_P,$$

where $id \in [0, n_{patches}-1]$ is index of the patch in one-dimensional array of patch-quads (assigned to each patch automatically when it enters the graphics pipeline).

3. If $B_P[i_P, j_P, k_P]=0$, then $(i_P, j_P, k_P)th$ patch is not processed, and we exit the algorithm (to do it, we set the numbers l_W and l_H to 0).

4. Calculate the numbers l_W and l_H of sub squares for the patch:

$$l_W = \max\left(\min\left(l, n-(l+1)j_P-1\right),1\right), \quad l_H = \max\left(\min\left(l, m-(l+1)i_P-1\right),1\right).$$

5. Subdivide the patch into a group of vertices of size $(l_W+1)\times(l_H+1)$ using primitive generator.

6. Transfer from TC-shader to TE-shader indices (i_P, j_P, k_P) of the patch, as well as the numbers
l_W and l_H of sub squares.

7. Calculate indices i_V and j_V of vertex in vertex (thread) group generated:

$$i_V = \lfloor l_H v + \varepsilon \rfloor, \quad j_V = \lfloor l_W u + \varepsilon \rfloor,$$

where (u,v) are normalized real coordinates of vertex in original patch-quad, $\varepsilon = 0.01$ is a small constant which compensates machine error of real numbers representation.

8. Calculate indices (i,j,k) of the voxel in 3D-voxel array:

$$i = (l+1)i_P + i_V, \quad j = (l+1)j_P + j_V, \quad k = k_P.$$

End of algorithm.

The three indices (i, j, k), calculated in TE-shader, unambiguously identify the voxel that will be visualized in the current voxel thread. The voxel is assumed to have unit dimensions. For such voxel, the developed G-shader constructs a polygonal model (strip) of a "trefoil", using the following

Algorithm of construction of visible "trefoil"

1. Transfer to G-shader the following parameters:
    - (i, j, k) - the indices of the voxel in 3D-voxel array (from TE-shader);
    - $n_{vcs,0},...,n_{vcs,5}$ - the coordinates of normals to voxel cube faces in VCS system;
    - M_{mv} - the current ModelView matrix;
    - M_{proj} - the projection matrix;
2. If $B[i, j, k] = 0$, then $(i, j, k)th$ voxel is not visualized, and we exit the algorithm.
3. Write positions $P_{ocs,0},...,P_{ocs,7}$ of all vertices of the cube (voxel) in OCS system:

$x_0 = j, \quad y_0 = i, \quad z_0 = k, \quad x_1 = j+1, \quad y_1 = i+1, \quad z_1 = k+1;$

$P_{ocs,0} = (x_0, y_0, z_0), \quad P_{ocs,1} = (x_1, y_0, z_0), \quad P_{ocs,2} = (x_0, y_1, z_0), \quad P_{ocs,3} = (x_1, y_1, z_0),$

$P_{ocs,4} = (x_0, y_0, z_1), \quad P_{ocs,5} = (x_1, y_0, z_1), \quad P_{ocs,6} = (x_0, y_1, z_1), \quad P_{ocs,7} = (x_1, y_1, z_1).$

4. Calculate positions $P_{vcs,0},...,P_{vcs,7}$ and $P_{ccs,0},...,P_{ccs,7}$ of all vertices of the cube in VCS system and in Clipping Coordinate System (CCS, [11]) respectively:

    Loop by v from 0 to 7

$P_{vcs,v} = M_{mv} P_{ocs,v}, \quad P_{ccs,v} = M_{proj} P_{vcs,v}.$

5. Calculate the number t of visible "trefoil" according to Eq.(1).
6. Set positions $P'_{ccs,0},...,P'_{ccs,7}$ of vertices of the strip of the tth "trefoil" (see Section 3) in OCS system, as well as positions $P'_{vcs,0},...,P'_{vcs,7}$ and coordinates $n'_{vcs,0},...,n'_{vcs,9}$ of vertex normals of this strip in VCS system (we use it for simplifying illumination calculation in fragment shader):

    Loop by s from 0 to 9

Get the number $a = N_t[s]$, where N_t is an array of numbers of normals (see Section 3).

Set $n'_{vcs,s} = n_{vcs,a}$.

Get the number $b = S_t[s]$, where S_t is an vertex index array (see Section 3).

Set $P'_{ccs,s} = P_{ccs,b}$, $P'_{vcs,s} = P_{vcs,b}$.

    End of loop.

7. Get the color $C = A[I_{i,j,k}]$ of visible "trefoil".
8. Construct visible "trefoil" by generating vertices of strip S_t with positions $P'_{ccs,0},...,P'_{ccs,7}$, normals $n'_{vcs,0},...,n'_{vcs,9}$ and color C.

End of algorithm.

The trefoil produced by G-shader is rasterized (a fixed stage of the graphics pipeline), resulting into pixels (fragments) forming the image of the trefoil. Pixel colors are calculated in parallel, independently of each other using developed fragment shader based on the Phong illumination model [11] with directional light source.

5. Results

Based on proposed methods and algorithms, a program module for real time visualization of digital core material model was developed. 3D visualization of allocated volume of DCM and construction of porosity plot are implemented in the module. Options to change the sampling range of absorption coefficient values, as well as the size of allocated volume and its offset inside the digital core material model, are also provided. During the visualization the researcher may rotate, zoom in and out the allocated volume. The module is added to program complex «CoreSimulator» [9], developed in FSI SRISA RAS, which is designed for various researches of DCM.

The program module was tested on personal computer equipped with NVidia GeForce GTX 1080 Ti graphics card (3584 computing cores) at Full HD and Ultra HD. For the testing digital core material models of sandstone ($2600 \times 2600 \times 7360$) and Bazhenov formation ($2560 \times 2560 \times 8600$) were used. Based on these data, a series of experiments, dealt with visualization of allocated volume from 100^3 to 1000^3 cells, corresponding to pore space and mineral skeleton, was carried out. Figures 3 and 4 demonstrate example frames of visualization of mesoporous (according to the classification used in [12]) core material of sandstone and poorly porous part of Bazhenov formation with granular dense inclusions. In all experiments, real-time mode was maintained when visualizing up to 3,9 million visible voxels (about 23,5 million triangles).

Also comparing video memory consumption (see Fig. 5) for constructing and visualization of the allocated volume between the solution developed, method basing only on geometry shader (GS) and method using fixed function pipeline (FFP) were provided. As a threshold for comparing video memory consumption, the amount of dedicated video memory installed on hi-end commodity graphics cards was chosen. The plot on Figure 5 demonstrates a significant advantage of our solution over GS and FFP methods, and, in contrast to them, the ability to work with allocated volumes of 2000^3 cells.

A comparison of time costs of reading the data from video memory, needed to render a single frame, is illustrated by the plot on Figure 6. The evaluation was being carried out based on average video memory bandwidth of 500 Gb/s and with time limit of 40 ms (corresponding to visualization rate of 25 frame/s). As seen from the plot, the reduced number of vertex invocations, provided by our solution, allows the time costs to be significantly decreased comparing to GS method. This opens an opportunity to treat allocated volumes of sizes more than 1000^3 cells. Real time visualization of such volumes is a promising subject of future researches.

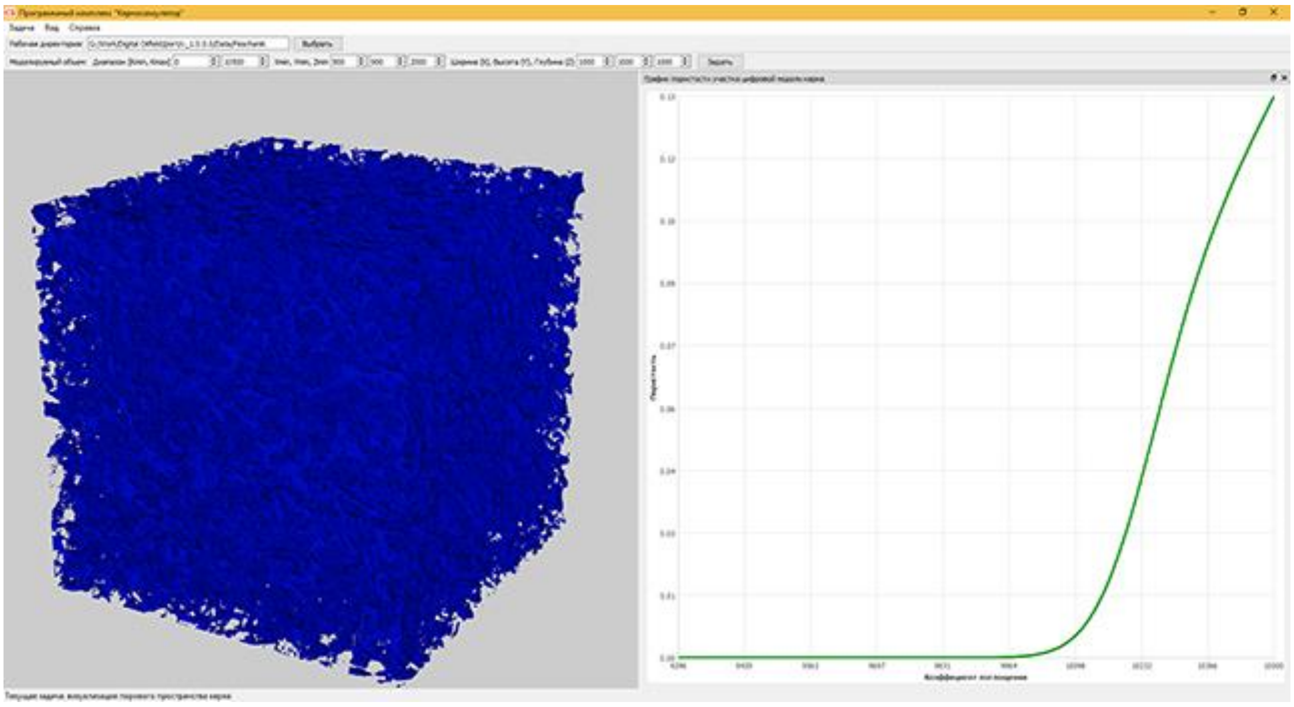
Fig. 3. A mesoporous part of sandstone DCM.

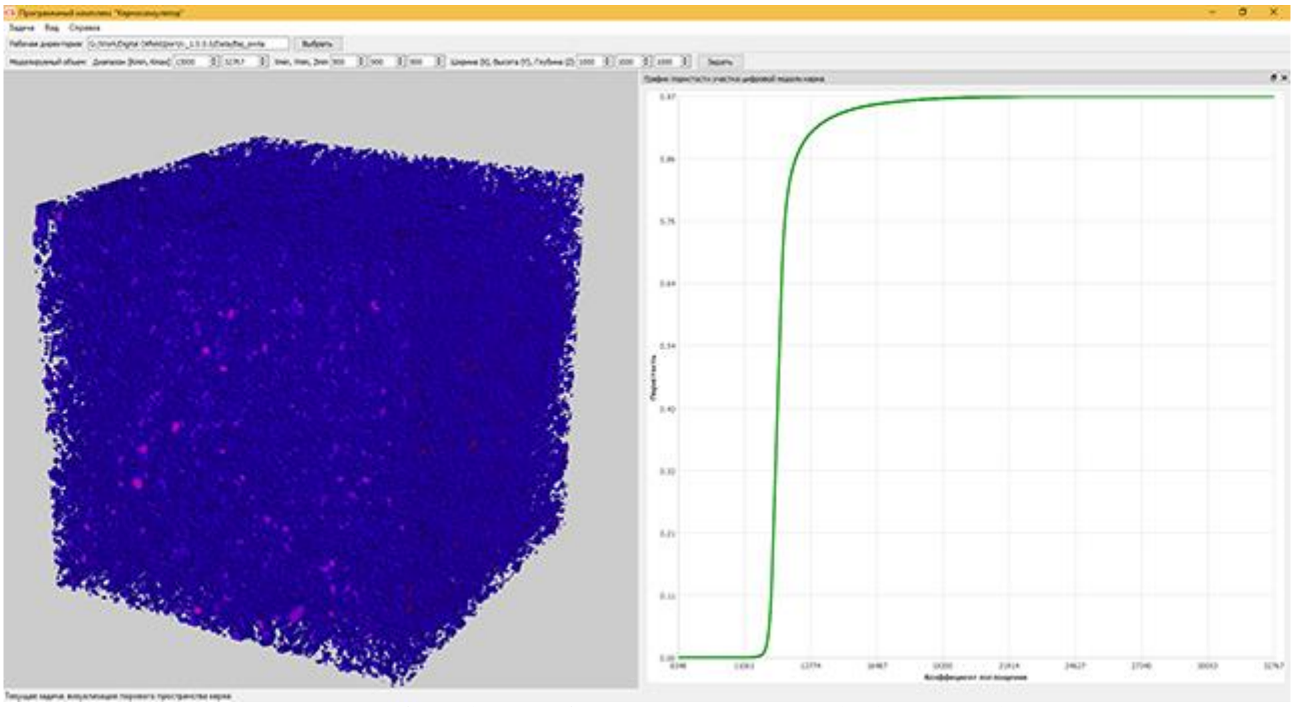
Fig. 4. A part of Bazhenov formation DCM with dense inclusions.

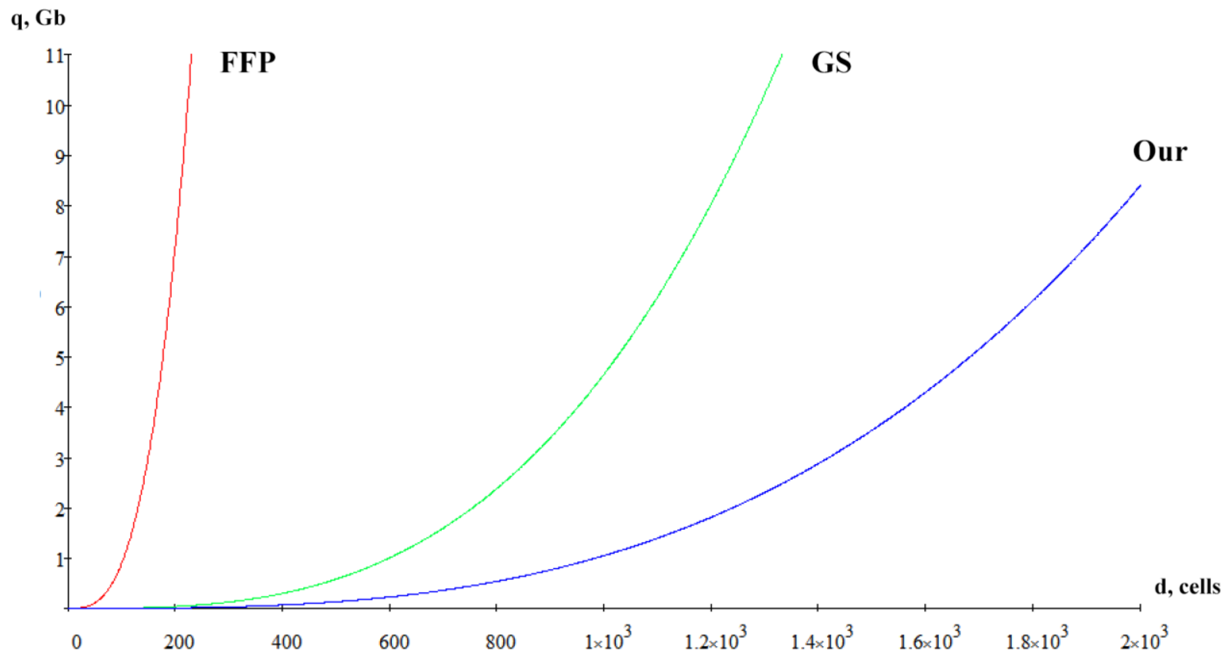

Fig. 5. Comparing video memory consumption q (in Gb) to construct a volume of d^3 cells size by means of FFP, GS methods and proposed solution (Our).

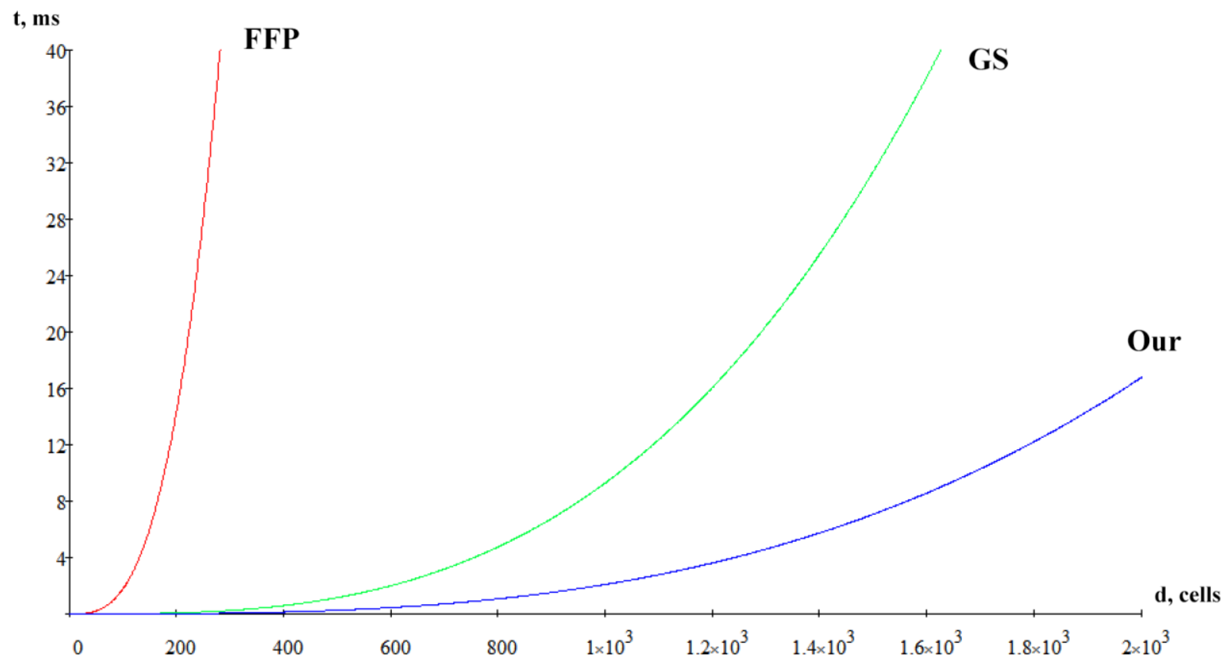

Fig. 6. Comparing time costs t (in ms) to transfer data from video memory to GPU for rendering a volume of d^3 cells size by means of FFP, GS methods and proposed solution (Our).

6. Acknowledgements

The publication is made within the state task on carrying out basic scientific researches (GP 14) on topic (project) “34.9. Virtual environment systems: technologies, methods and algorithms of mathematical modeling and visualization” (0065-2019-0012).

References

1. Betelin V. B., Smirnov N.N., Stamov L.I., Skryleva E.I. Developing the structure of core pores based on processing of tomography data // Proceedings in Cybernetics. – 2018. – № 2 (30). – pp. 87-92. [in Russian] (https://drive.google.com/file/d/1tc7rJmObCV132tetV1j6AZfPULeA50Bl/view).
2. Bondarev A.E., Bondarenko A.V., Galaktionov V.A. Visual analysis procedures for multi-dimensional data // Scientific Visualization, Vol. 10, № 4, 2018, pp. 120-133 (doi: 10.26583/sv.10.4.09) (http://sv-journal.org/2018-4/09/index.php?lang=ru) [in Russian].
3. Digital Sandstone Rock Analysis Scanned with High-Resolution X-ray Computed Tomography // General Electric Oil & Gas Digital Solutions, 2014 (https://youtu.be/BgbPoovPYi8).
4. Kaufman A., Cohen D., Yagel R. Volume graphics // Computer. – 1993. – Vol. 26, № 7. - pp. 51–64.
5. Crassin C., Neyret F., Lefebvre S., Eisemann E. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering // Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, ACM, 2009, PP. 15-22.
6. ParaView (https://www.paraview.org/Wiki/ParaView).
7. OpenGL Minecraft Style Volume Rendering (http://bytebash.com/2012/03/opengl-volume-rendering/).
8. Jabłoński S., Martyn T. Real-time voxel rendering algorithm based on Screen Space Billboard Voxel Buffer with Sparse Lookup Textures // In Proc. of WSCG - 24th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision 2016, pp. 27-36 (https://otik.zcu.cz/bitstream/11025/29528/1/Jablonsky.pdf).
9. Timokhin P. Yu., Mikhaylyuk M. V. Implementation technology of multitasking graphical shell of visualization system for digital model of the core material // Proceedings in Cybernetics. – 2018. – № 3 (31). – pp. 247-254. [in Russian] (https://drive.google.com/file/d/1O3HtX518kfywQLnfObLWTVIzbm3swnh6/view).
10. Nießner M., Keinert B., Fisher M., Stamminger M., Loop C., Schäfer H. Real-Time Rendering Techniques with Hardware Tessellation // Computer Graphics Forum, Vol. 35 (1), 2016, pp. 113-137 (doi: 10.1111/cgf.12714) (http://graphics.stanford.edu/~mdfisher/papers/realtimeRendering.pdf).
11. Bailey M., Cunningham S. Graphics Shaders: Theory and Practice, Second Edition // CRC Press. – 2011. – 518 p.
12. Hanin A. A. Porody-kollektory nefti i gaza i ih izuchenie [Oil and gas reservoir rocks and their study]. // Moscow: Nedra, 1969 [in Russian].