ISSN 0361-7688, Programming and Computer Software, 2020, Vol. 46, No. 3, pp. 244–249. © Pleiades Publishing, Ltd., 2020.
Russian Text © The Author(s), 2020, published in Programmirovanie, 2020, Vol. 46, No. 3.

Method to Extract Isosurfaces on the GPU by Means of Programmable Tessellation

P. Y. Timokhin[a],* and M. V. Mikhaylyuk[a],**

[a] *Scientific Research Institute for System Analysis, Russian Academy of Sciences, Nakhimovskii pr. 36/1, Moscow, 117218 Russia*

**e-mail: webpismo@yahoo.de*

***e-mail: mix@niisi.ras.ru*

Received December 20, 2019; revised January 10, 2020; accepted January 18, 2020

Abstract—This paper considers the task of real-time visualization of detailed 3D scalar fields through isosurfaces (surfaces representing constant values of scalar fields). A new method is proposed to overcome the limitations associated with the intensive reading of dummy (non-visualized) vertices from video memory and overheads of their storage, which arise when implementing isosurface polygonization methods on the GPU. The proposed method is based on the efficient generation of GPU threads (in which an isosurface model is constructed) by the programmed tessellation of quadrangular patches into regular vertex grids. We also propose a modified technology for marching cubes implementation on the GPU that is based on the developed method and allows the time cost of GPU thread generation and video memory footprint to be significantly reduced. Based on the proposed solutions, a software complex for the real-time construction and visualization of polygonal models of isosurfaces is implemented and tested, as well as the verification of synthesized images is carried out.

DOI: 10.1134/S036176882003010X

1. INTRODUCTION

Currently, many important scientific and applied fields intensively employ the visualization of 3D scalar fields obtained by numerical modeling or scanning objects (processes) with complex internal structures. In this regard, the real-time visualization [1] (with an image synthesis frequency of at least 25 images per second) of the surface representing the constant value of a scalar field (the so-called isosurface) is one of the effective methods. It can be used for skeleton visualization in computed tomography [2], phase boundary visualization in digital models of core material samples [3], 3D models of terrain with negative slopes in virtual environment systems [4], etc. The extraction of the isosurface from the field is based on the computation of the points with a constant field value in the cells of a scalar grid. The more complex the object under study, the more detailed scalar grid is required for its adequate representation and, therefore, the more computations must be carried out to extract the isosurface. In the modern context of ultra-high-definition screens and stereo devices [5], the amount and time of these graphics computations increase severalfold, which complicates the real-time synthesis of isosurface images. Thus, it is required to develop efficient methods and algorithms for isosurface extraction based on parallelization of computations on modern thousand-core GPUs and technologies for parallel processing of graphics data.

In this paper, we propose a new method for solving this task on the GPU based on scalar field polygonization (extracting a set of triangular faces that approximates the isosurface) that is carried out in real time in parallel GPU threads generated using programmed tessellation of parametric graphics primitives [6]. The proposed solution is implemented in C++ and GLSL (shader programming language) by using the OpenGL graphics library.

2. APPROACHES TO ISOSURFACE EXTRACTION

There are two main directions for the development of isosurface extraction methods.

The first direction includes methods based on ray casting from the observer's position through the pixels of the screen up to the intersection with the isosurface [7–9]. This approach yields high-quality isosurface images. However, the rate of synthesizing these images decreases significantly with increasing the number of pixels (rays) to be processed, which complicates real-time visualization on ultra-high-definition screens. The latest studies [10] suggest that the recently introduced hardware support of ray tracing on NVidia RTX graphics cards can significantly improve this situation.

The second direction includes methods based on the polygonal approximation of the isosurface in the cells of a 3D grid (render grid) superimposed onto a scalar field grid [11–19]. With this approach, active (intersected by the isosurface) edges and cells of the render grid are found; then, for these edges and cells, polygons of the isosurface are generated in accordance with certain rules. There are direct [11–13], dual [14–16], and hybrid [17–19] methods for isosurface polygonization. With direct methods, in each active cell, polygons whose vertices lie on the active edges of the cell are created (a classic example is marching cubes [11]). With dual methods, polygons are created for active edges, while the vertices of these polygons are placed in active cells. Hybrid methods combine direct and dual methods and act as dual ones when active cells contain isosurface features, e.g., sharp edges and cone-shaped vertices. An extensive investigation and comparison of isosurface polygonization methods can be found in [20].

With the addition of the programmable (shader) stages to the graphics pipeline, the research direction associated with the parallel GPU-based processing of active cells and edges became very popular [21–27]. The best results were obtained when parallelizing direct methods whereby, in contrast to dual methods, each active cell is triangulated independently. The efficiency of the GPU-based versions of direct methods depends heavily on the time cost of generating and executing GPU threads, as well as on the size of video memory footprint. In the era of thousand-core GPUs with hierarchical video memory, the bottleneck of visualizing high-poly models is the number of accesses to global video memory (the largest and slowest part of VRAM). The number of these memory accesses can be reduced by constructing polygons for an active cell in the geometry shader (one of the programmable stages of the graphics pipeline). To run a GPU thread with the geometry shader, it is sufficient to send a vertex to its input. A common approach is to create a vertex buffer object (VBO) with dummy vertices (vertex per render grid cell) in video memory and send this array of vertices to the graphics pipeline. Once the dummy vertices arrive at the GPU cores, on each core, the geometry shader replaces each vertex (independently and in parallel) with a certain polygonal structure if the cell is active. Even though the polygonal model of the isosurface is in fact constructed on the GPU, this approach has some significant limitations associated with the intensive reading of the vertex data from global video memory and additional overheads of their storage. These limitations prevent the increase in the size of the render grid and hinder the real-time construction of isosurfaces for complex objects.

In this paper, we propose a new method to overcome these limitations; it is based on two new programmable stages—tessellation control shader and tessellation evaluation shader—used to construct a polygonal model of the isosurface [6]. These stages precede the geometry shader and, in the general case, are employed for the parallel GPU-based generation of a large set of connected triangular approximations from parametric graphics primitives (patches), e.g., to construct a polygonal model of the Earth's surface [28]. In this work, we use a special case of programmable tessellation whereby a regular grid of vertices is generated from a quadrangular patch (quad patch) on the GPU. By the example of constructing a polygonal model of an isosurface based on marching cubes, we demonstrate how this approach can significantly reduce the time required for GPU thread generation and lower video memory footprint.

3. TECHNOLOGY TO IMPLEMENT MARCHING CUBES ON THE GPU USING TESSELLATION SHADERS

Suppose that we have an $m \times n \times q$ render grid R each node (vertex) of which corresponds to a certain value S of a scalar field under study. The task consists in constructing a polygonal model of a surface representing a constant value S^* in the active cells of the grid R. Each cell for all eight vertices of which the condition $S < S^*$ or $S \geq S^*$ does not hold is considered active, i.e., the cells that lie entirely inside or outside the isosurface are excluded.

Let us consider the $R_{i,j,k}$th cell. We enumerate the vertices and edges of this cell with integers from 0 to 7 and from 0 to 11, respectively. A unit bit is assigned to each vertex satisfying $S < S^*$; otherwise, a zero bit is assigned (see Fig. 1). According to the rule of marching cubes, a vertex of a triangle (polygon) is placed on an edge whose vertices have different bits (active edge); the coordinates P of this vertex are found as follows:

$$P = P_a + \frac{S^* - S_a}{S_b - S_a}(P_b - P_a), \quad (1)$$

where P_a and P_b are coordinates for the vertices of the active edge, while S_a and S_b are the values of the scalar field at the vertices of the active edge. By K we denote a configuration of eight consecutive bits. Each value K (from 0 to 255) is uniquely associated with a list of triangles, representing a variant of a polygonal part of the isosurface model. Each such list includes 0 to 5 triangles and is defined by a sequence of 16 indices: three active edge indices per triangle and (–1) filling the sequence to the end. All 256 sequences constitute an array T of triangle lists (see [12]).

The list of triangles in each $R_{i,\,j,\,k}$th active cell is constructed in an individual GPU thread (marching cube thread). In this paper, these threads are generated using programmed tessellation of quad patches. Having properly programmed the graphics pipeline, each quad patch can be subdivided in parallel into a regular grid of up to $D \times D$ vertices, where $D = L_{\max} + 1$ and $L_{\max}$ is the maximum tessellation level (the maximum

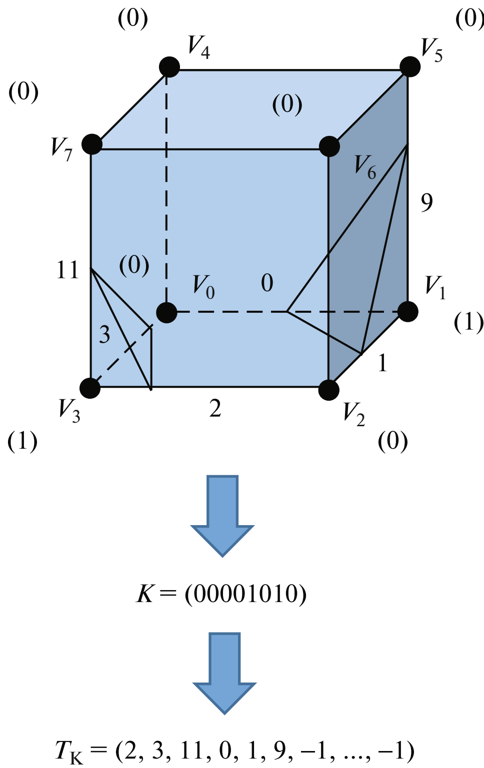

Fig. 1. Encoding a cell of the render grid.

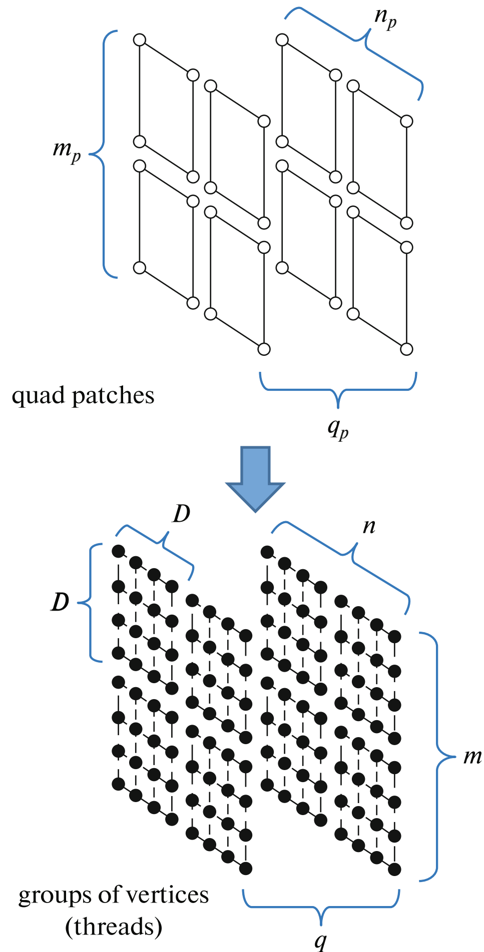

Fig. 2. Generation of marching cube threads.

number of segments into which the side of a quad patch can be subdivided). In OpenGL 4.0, this number is at least 64. In accordance with the architecture of the graphics pipeline, each vertex obtained as a result of tessellation is processed in an individual GPU thread that we program to construct a list of triangles corresponding to the cell (as part of the isosurface). In contrast to the solutions based on the CUDA architecture, an important advantage of the proposed approach is that these vertices (GPU threads) are generated directly on the GPU without wasting the video memory resource while remaining in the framework of the graphics pipeline.

The proposed technology for the GPU-based implementation of marching cubes comprises two stages. At the first stage (loading the source data), a floating-point 3D texture with the values S of render grid vertices, an integer 2D texture that encodes the array T of triangle lists, and a VBO that contains $m_p n_p q_p$ quad patches, where $m_p = \lceil m/D \rceil$, $n_p = \lceil n/D \rceil$, and $q_p = q$ (see Fig. 2), are loaded into video memory. At the second stage (visualization), the quad patches from the VBO are sent to the graphics pipeline, where they are distributed among the GPU cores and are processed in parallel using the tessellation control shader (TCS), tessellation evaluation shader (TES), geometry shader (GS), and fragment shader (FS). Below, we consider the operation of these shaders in more detail.

TCS shader. At this stage, two sets of parameters are evaluated. The first set is a triplet of indices (i_p, j_p, k_p) for the row, column, and slice of the patch processed by the TCS shader in the 3D array of quad patches (see Fig. 2). These parameters are obligatory to associate the GPU threads with the cells of the render grid. The triplet of indices (i_p, j_p, k_p) is computed based on the built-in variable gl_PrimitiveID, which is a running number g of the patch that takes values on the interval $[0, m_p n_p q_p - 1]$:

$$k_p = \lfloor g/N \rfloor, \quad i_p = \left\lfloor \frac{g - Nk_p}{M} \right\rfloor,$$
$$j_p = g - Nk_p - Mi_p.$$

where $N = m_p n_p$ and $M = Nq_p$. The second set includes parameters that specify the width and height of the 2D grid of vertices (group of marching cube threads). These width and height are determined by the levels l_w and l_h of quad patch tessellation, respectively, which are computed as follows:

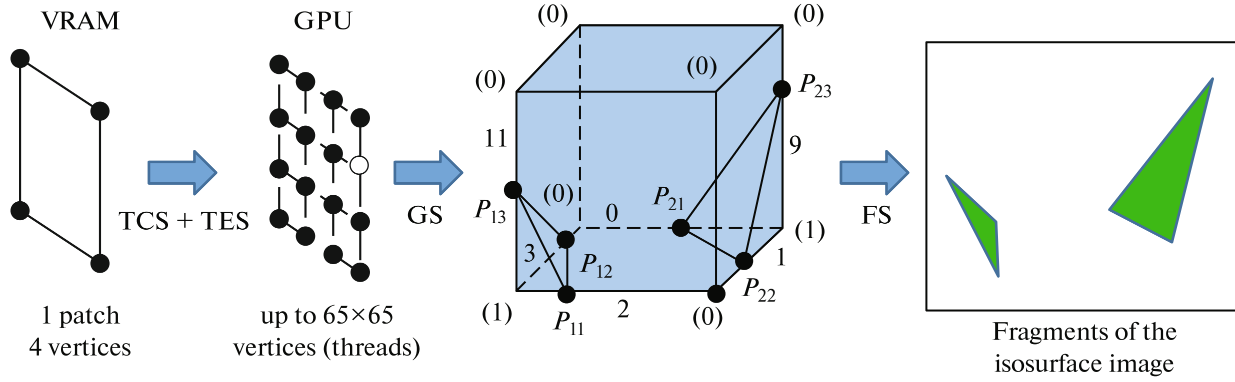

Fig. 3. Scheme of the isosurface visualization pipeline.

$$l_w = \max(\min(L_{\max}, n - Dj_p - 1), 1),$$
$$l_h = \max(\min(L_{\max}, m - Di_p - 1), 1).$$

TES shader. This shader processes the vertices of the 2D vertex grid obtained by the tessellation of the quad patch in parallel while associating each vertex (thread) with a render grid cell. This is done by assigning the triplet of attributes (i_c, j_c, k_c), which are the row, column, and layer indices of the corresponding cell in the render grid, to the vertex under processing. These attributes are computed as follows:

$$i_c = Di_p + i, \quad j_c = Dj_p + j, \quad k_c = k_p,$$

where i and j are the row and column indices of the vertex in the 2D vertex grid obtained by the patch tessellation:

$$i = \lfloor l_h v + \varepsilon \rfloor, \quad j = \lfloor l_w u + \varepsilon \rfloor,$$

where $(u, v) \in [0,1]$ are the normalized floating-point coordinates of the vertex in the 2D vertex grid (which are computed automatically in the process of tessellation in the TES shader) and ε is a small constant used to compensate for the machine error in the representation of real numbers.

GS shader. This shader receives a triplet of indices (i_c, j_c, k_c) for the render grid cell from the TES shader and performs parallel processing of all these cells. The processing begins with computing the configuration number K_{i_c,j_c,k_c} for the (i_c, j_c, k_c)th cell:

$$K_{i_c,j_c,k_c} = \sum_{p=0}^{7} 2^p b_p,$$

where p is the running number of a vertex in the (i_c, j_c, k_c)th cell (see Fig. 1); the flag b_p is 1 if $S_p < S^*$ and is 0 otherwise (S_p is the value of the scalar field at the pth vertex of the cell). If $K_{i_c,j_c,k_c} = 0$ or $K_{i_c,j_c,k_c} = 255$ (the cell lies entirely inside or outside the isosurface), then the GS shader terminates without generating any polygonal geometry in the cell. Otherwise, (a) based on the configuration number K_{i_c,j_c,k_c}, the lists of edges that form the polygons of the isosurface in the (i_c, j_c, k_c)th cell are extracted from the texture T of triangle lists; (b) from the 3D texture of the render grid, the values S for the vertices of the edges are extracted and the coordinates of the points on these edges are computed by Eq. (1); and (c) based on the points obtained, the triangles of the isosurface in the (i_c, j_c, k_c)th cell are constructed in accordance with the sequence order of the edges in the extracted list. The implementation of triangle emission in the GS can be found in [29].

FS shader. The triangles constructed by the GS shader are rasterized (fixed stage of the graphics pipeline) to be converted into fragments of the isosurface image. The FS shader computes the colors of the resulting fragments independently and in parallel, based on the Phong lighting model [29] with a directional light source.

Figure 3 shows the general scheme of the visualization pipeline developed. Once all quad patches pass through this pipeline, the resulting image of the extracted isosurface is synthesized in the frame buffer.

4. RESULTS

Based on the proposed technology, a software complex was created that implements the extraction of polygonal models of isosurfaces by means of programmable tessellation. To test this complex, we used 3D scalar fields of water saturation in a porous sample of oil-bearing rock, which were obtained by numerical modeling on a computational grid of 100^3 cells. Figure 4 shows the extracted surface that corresponds to 35% saturation level. The polygonal model of the isosurface was constructed and visualized at a resolution of 3840 × 2160 on GeForce GTX 1080 Ti (3584 cores, 11 GB VRAM); the average frequency of visualization was approximately 100 frames per second. The isosurface obtained was verified by extracting an isosurface from the same array of scalar data in MATLAB [30].

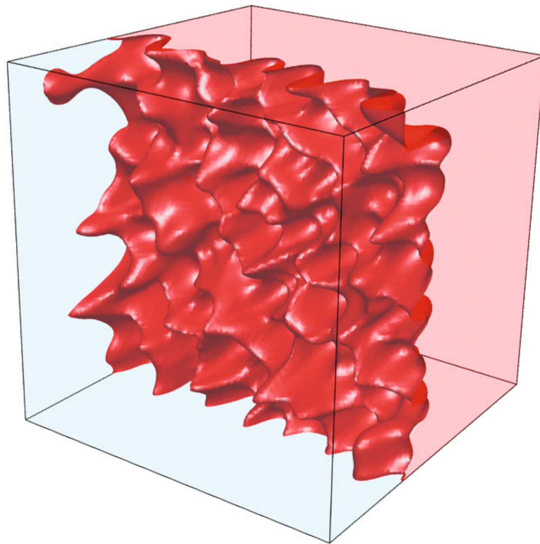

Fig. 4. Example of isosurface visualization.

In addition, for the proposed solution, we estimated video memory footprint (in GB) and time it takes to generate GPU threads (in milliseconds) with increasing size of the render grid up to 2000^3 cells. Figure 5 shows the consumption of time (T) and video memory (M) for (a) the proposed implementation and (b) the implementation of the common approach described in Section 2. As compared to the implementation (b), the proposed solution reduces the time required for GPU thread generation by a factor of five and video memory footprint by a factor of eight. Based on these results, in our future works, we intend to conduct a research on the real-time extraction of isosurfaces from extended 3D scalar fields (construction of a extended terrain with negative slopes).

5. CONCLUSIONS

In this paper, we have considered the task of real-time visualization of detailed 3D scalar fields through isosurfaces. To overcome the revealed limitations, which hinder the real-time construction of polygonal models of isosurfaces for complex objects, a new GPU-based method for isosurface extraction using the tessellation control shader and tessellation evaluation shader has been proposed. An efficient technology for GPU-based implementation of marching cubes has been proposed. It is based on the developed method and allows the time cost of GPU thread generation and video memory footprint to be significantly reduced as compared to the GPU-based implementation of marching cubes without tessellation. The proposed solutions have been implemented in a software complex for the real-time construction and visualization of polygonal models of isosurfaces. The developed complex has been successfully tested on 3D scalar fields of water saturation in a sample of oil-bearing rock, which were obtained by numerical modeling. The resulting isosurface images have been verified using MATLAB. In the future, we intend to further develop the proposed technology to enable the construction and visualization of terrain models with negative slopes.

FUNDING

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).

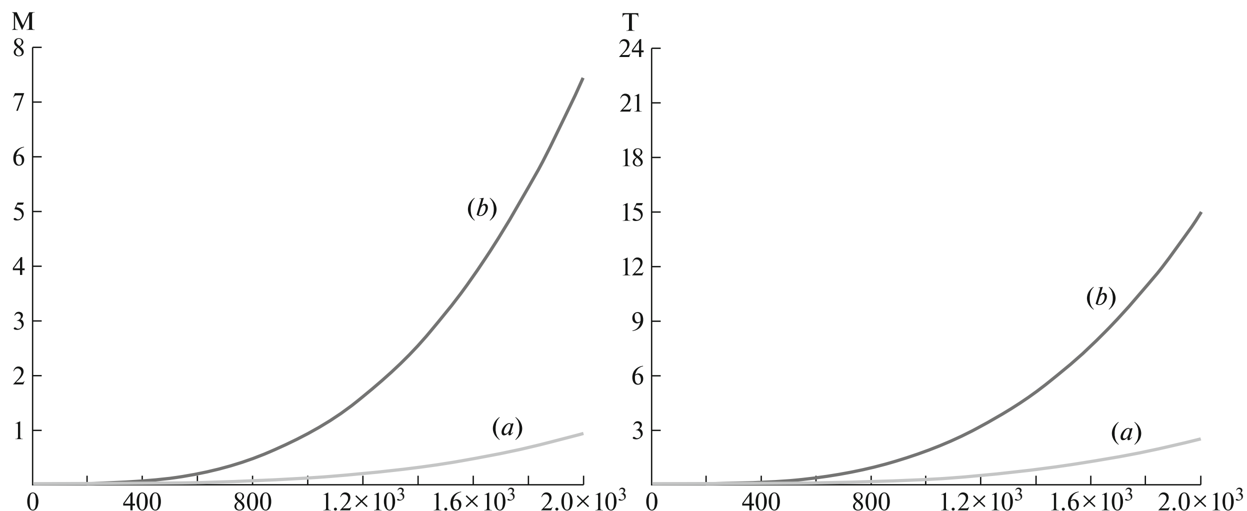

Fig. 5. Comparison between (a) proposed solution and (b) GPU-based implementation without the tessellation shaders in terms of video memory footprint (M) and GPU thread generation time (T).

REFERENCES

1. Barladian, B.Kh., Voloboy, A.G., Galaktionov, V.A., Knyaz', V.V., Koverninskii, I.V., Solodelov, Yu.A., Frolov, V.A., and Shapiro, L.Z., Efficient implementation of OpenGL SC for avionics embedded systems, *Program. Comput. Software,* 2018, vol. 44, pp. 207–212.
2. Gavrilov, N. and Turlapov, V., General implementation aspects of the GPU-based volume rendering algorithm, *Sci. Visualization,* 2011, vol. 3, no. 1, pp. 19–31. http://sv-journal.org/2011-1/02/index.html.
3. Timokhin, P. and Mikhaylyuk, M., Compact GPU-based visualization method for high-resolution resulting data of unstable oil displacement simulation, *Proc. 29th Int. Conf. Computer Graphics and Vision (GraphiCon),* Bryansk, 2019, vol. 2485, pp. 4–6.
4. Shakaev, V., Polygonizing volumetric terrains with sharp features, *Tr. 26-i Mezhdunar. Nauchn. Konf. GraphiCon* (Proc. 26th Int. Sci. Conf. GraphiCon), Nizhny Novgorod, 2016, pp. 364–368.
5. Timokhin, P.Yu., Mikhailyuk, M.V., Vozhegov, E.M., and Panteley, K.D., Technology and methods for deferred synthesis of 4K stereo clips for complex dynamic virtual scenes, *Tr. Inst. Sistemnogo Program. Ross. Akad. Nauk* (Proc. Inst. Syst. Program. Russ. Acad. Sci.), 2019, vol. 31, no. 4, pp. 61–72.
6. Segal, M. and Akeley, K., The OpenGL graphics system: A Specification, Version 4.6, Core profile, 2006–2018. https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.
7. Parker, S., Shirley, P., Livnat, Y., et al., Interactive ray tracing for isosurface rendering, *Proc. IEEE Visualization (VIZ),* 1998, pp. 233–238.
8. Hadwiger, M., Sigg, C., Scharsach, H., Buhler, K., and Gross, M., Real-time ray-casting and advanced shading of discrete isosurfaces, *Comput. Graphics Forum,* 2005, vol. 24, no. 3, pp. 303–312.
9. Kim, M., GPU isosurface raycasting of FCC datasets, *Graphical Models,* 2013, vol. 75, no. 2, pp. 90–101.
10. Sanzharov, V.V., Gorbonosov, A.I., Frolov, V.A., and Voloboy, A.G., Examination of the Nvidia RTX, *Proc. 29th Int. Conf. Computer Graphics and Vision (GraphiCon),* 2019, vol. 2485, pp. 7–12.
11. Lorensen, W.E. and Cline, H.E., Marching cubes: A high resolution 3D surface construction algorithm, *Proc. 14th Annu. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH),* Anaheim, 1987, vol. 21, no. 4, pp. 163–169.
12. Bourke, P., Polygonising a scalar field, 1994. http://paul-bourke.net/geometry/polygonise.
13. Kobbelt, L., Botsch, M., Schwanecke, U., and Seidel, P., Feature sensitive surface extraction from volume data, *Proc. 28th Annu. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH),* 2001, pp. 57–66.
14. Ju, T., Losasso, F., Schaefer, S., and Warren, J., Dual contouring of Hermite data, *Proc. ACM Trans. Graphics (TOG),* 2002, vol. 21, no. 3, pp. 339–346.
15. Schmitz, L., Dietrich, C., and Comba, J., Efficient and high quality contouring of isosurfaces on uniform grids, *Comput. Graphics Image Process.,* 2009, pp. 64–71.
16. Nielson, G., Dual marching cubes, *IEEE Visualization,* 2004, pp. 489–496.
17. Schaefer, S. and Warren, J., Dual marching cubes: Primal contouring of dual grids, *Comput. Graphics Forum,* 2005, vol. 24, no. 2, pp. 195–201.
18. Ho, C.-C., Wu, F.-C., Chen, B.-Y., Chuang, Y.-Y., and Ouhyoung, M., Cubical marching squares: Adaptive feature preserving surface extraction from volume data, *Proc. Eurographics,* 2005, vol. 24, no. 3, pp. 537–545.
19. Manson, J. and Schaefer, S., Isosurfaces over simplicial partitions of multiresolution grids, *Comput. Graphics Forum (Proc. Eurographics),* 2010, vol. 29, no. 2, pp. 377–385.
20. De Araújo, B.R., Lopes, D.S., Jepp, P., Jorge, J.A., and Wyvill, B., A survey on implicit surface polygonization, *ACM Computing Surveys,* 2015, vol. 47, no. 4, pp. 1–39.
21. Matsumura, M. and Anjo, K., Accelerated isosurface polygonization for dynamic volume data using programmable graphics hardware, *Proc. SPIE-IS&T Electronic Imaging, Visualization and Data Analysis,* 2003, vol. 9, pp. 145–152.
22. *Visualization Handbook,* Hansen, C. and Johnson, C., Eds., Elsevier, 2004.
23. Goetz, F., Junklewitz, T., and Domik, G., Real-time marching cubes on the vertex shader, *Proc. Eurographics,* 2005, pp. 1–4.
24. Tatarchuk, N., Shopf, J., and DeCoro, C., Real-time isosurface extraction using the GPU programmable geometry pipeline, *Proc. ACM SIGGRAPH,* 2007, pp. 122–137.
25. Dyken, C., Ziegler, G., Theobalt, C., and Seidel, H.-P., High-speed marching cubes using histopyramids, *Comput. Graphics Forum,* 2008, vol. 27, no. 8, pp. 2028–2039.
26. Akayev, A.A., Kuzin, A.K., Orlov, S.G., Chetverushkin, B.N., Shabrov, N.N., and Iakobovski, M.V., Generation of isosurface on a large mesh, *Proc. IASTED Int. Conf. Automation, Control, and Information Technology (ACIT),* 2010, pp. 236–240.
27. Chen, J., Jin, X., and Deng, Z., GPU-based polygonization and optimization for implicit surfaces, *Visual Comput.,* 2015, vol. 31, no. 2, pp. 119–130.
28. Mikhaylyuk, M.V., Timokhin, P.Y., and Maltsev, A.V., A method of Earth terrain tessellation on the GPU for space simulators, *Program. Comput. Software,* 2017, vol. 43, pp. 243–249.
29. Bailey, M. and Cunningham, S., *Graphics Shaders: Theory and Practice,* CRC Press, 2011, 2nd ed.
30. MATLAB documentation, Volume visualization. https://www.mathworks.com/help/matlab/ref/isosurface.html.

Translated by Yu. Kornienko