Effective technology to visualize virtual environment using 360-degree video based on cubemap projection

P.Y. Timokhin[1], M.V. Mikhaylyuk[1]
webpismo@yahoo.de | mix@niisi.ras.ru
[1]Federal State Institution «Scientific Research Institute for System Analysis of the Russian Academy of Sciences», Moscow, Russia

The paper dealt with the task of increasing the efficiency of high-quality visualization of virtual environment (VE), based on video with a 360-degree view, created using cubemap projection. Such a visualization needs VE images on cube faces to be of high resolution, which prevents a smooth change of frames. To solve this task, an effective technology to extract and visualize visible faces of the cube is proposed, which allows the amount of data sent to graphics card to be significantly reduced without any loss of visual quality. The paper proposes algorithms for extracting visible faces that take into account all possible cases of hitting / missing cube edges in the camera field of view. Based on the obtained technology and algorithms, a program complex was implemented, and it was tested on 360-video of a virtual experiment to observe the Earth from space. Testing confirmed the effectiveness of the developed technology and algorithms in solving the task. The results can be applied in various fields of scientific visualization, in the construction of virtual environment systems, video simulators, virtual laboratories, in educational applications, etc.

***Keywords**: scientific visualization, cubemap projection, 360 degree video, virtual environment.*

1. Introduction

An important part of many up-to-date researches is the visualization of scientific data and experiments using a 3D virtual environment (VE) that simulates the object of study [1]. This is especially demanded in the fields where research is associated with high risk and work in hard-to-reach environments: medicine [2], space [3], oil and gas industry [4], etc. One of the effective forms to share such visualization between researchers is video with a 360-degree view, while watching which one is able to rotate the camera in an arbitrary direction and feel the effect of immersion in VE. So, for instance, using 360-video, anyone can explore the virtual model of the center of the galaxy [5].

To create a 360-video, various methods are used to unwrap a spherical panorama onto a plane [6]: by means of equidistant cylindrical projection, cubemap projection, a projection on the faces of the viewer's frustum [7], etc. One of the widely distributed is cubemap projection in which the panorama is mapped to 6 faces of the cube, where each face covers a viewing angle of 90 degrees. When playing such a video, the viewer is inside the cube and looks at its faces with images of the virtual environment. In order to feel immersion effect, it is important that the images on the faces have a sufficiently high resolution, i.e. the viewer should not see their discrete (pixel) structure. Increasing of the resolution leads to the formation of a large stream of graphic data, which impedes the visualization process. In this regard, the task to reduce the amount of streaming data without noticeable loss of visualization quality is arisen.

In this paper, an effective technology for solving this task is proposed, which is based on the extraction and visualization of cube faces been visible towards the viewer. The technology is implemented in C ++ using the OpenGL graphics library.

2. The pipeline of 360-video visualization

Consider the task of visualization of 360-video with frames comprising images of cube faces (*face textures*) as shown in Fig. 1. The faces are named as they are seen by the viewer inside the cube. To visualize 360-video, a virtual 3D scene is created containing unit cube model centered at the origin of the World Coordinate System (WCS). Viewer's virtual camera C_V is placed in cube center and is initially directed to the front face (Fig. 2), where $\boldsymbol{v}$ and $\boldsymbol{u}$ are "view" and "up" vectors of camera C_V (in WCS), and $\boldsymbol{r} = \boldsymbol{v} \times \boldsymbol{u}$ is "right" vector. The pipeline of 360-video visualization includes reading a frame from video file with a frequency specified in video; extracting face textures from the frame and applying them to cube model; synthesizing the image of textured cube model from camera C_V. When watching a 360-video, we allow camera rotation around the X and Y axes of its local coordinate system, which corresponds to tilting the head up/down and left/right.

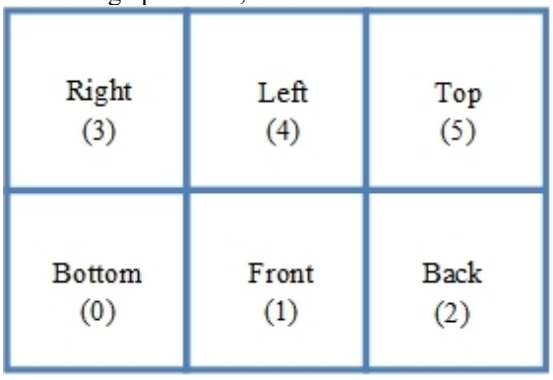


Fig. 1. The location of cube faces in the frame of 360-video

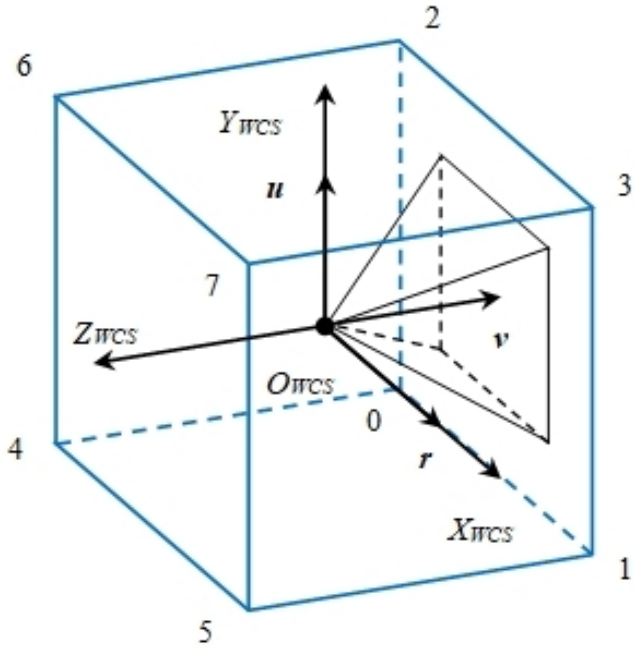


Fig. 2. Viewer's camera C_V and visualization cube

The bottleneck of the pipeline described is transferring of face textures from RAM to video memory (VRAM). Therefore, transferring of all 6 face textures (the entire frame of 360-video) to VRAM will be extremely inefficient and impede the smoothness of visualization. To solve this problem, we propose a technology based on the extraction and visualization of only those cube faces that are seen in the camera C_V.

3. The technology to extract and visualize visible cube faces

To identify visible cube faces, we introduce the term "*face pair*" - two cube faces with a common edge. We enumerate cube vertices, as shown in Fig. 2, and specify through them the edges: {0, 1}, {1, 5}, {5, 4}, {4, 0}, {6, 4}, {7, 5}, {3, 1}, {2, 0}, {2, 3}, {3, 7}, {7, 6}, {6, 2}. These edges are corresponded by the following face pairs: {0, 1}, {0, 3}, {0, 2}, {4, 0}, {2, 4}, {2, 3}, {3, 1}, {1, 4}, {1, 5}, {3, 5}, {5, 2}, {4, 5}, where 0…5 are face numbers from Fig. 1. Depending on orientation and projection parameters of camera C_V, cube edges may hit the frustum of the camera, or miss it. Every edge hitting the frustum determines face pair needed to render. No edges hitting the frustum means that camera C_V captures some single cube face.

The proposed technology includes five stages. At the ***first stage***, camera C_V's frustum parameters are determined. At the ***second stage***, boolean table H of cube vertices visibility is created. At the ***third stage***, visible face pairs are extracted using table H. At the ***fourth stage***, single visible face is extracted (if necessary). At the ***fifth stage***, extracted cube faces are visualized. Let's consider these stages in detail.

Frustum parameters

To determine visible cube faces, we need the following parameters of camera C_V's frustum: γ_{hor} and γ_{vert} - horizontal and vertical FOV (field of view) angles; d_n and d_f - distances to the near and far clipping planes. The angle $\gamma_{hor} \in [\delta, \pi - \delta]$ is user-defined, where δ is a small constant ($\delta = 1°$ in our work), and the angle γ_{vert} is determined by the ratio

$$\mathrm{tg}(\gamma_{vert}/2) = \mathrm{tg}(\gamma_{hor}/2)/aspect, \quad (1)$$

where $aspect \geq 1$ is the aspect ratio of camera C_V (the ratio of frame's width to its height). The distance to the far plane should not be less than half of the longest cube diagonal, so we take $d_f = \sqrt{3}/2 + \varepsilon$, where ε is machine error of real numbers representation. The near clipping plane should be located so that the near base of the frustum does not contact cube faces from the inside. Fig. 3 shows that such contact point is the intersection of the side line of camera C_V's FOV with the center of cube face.

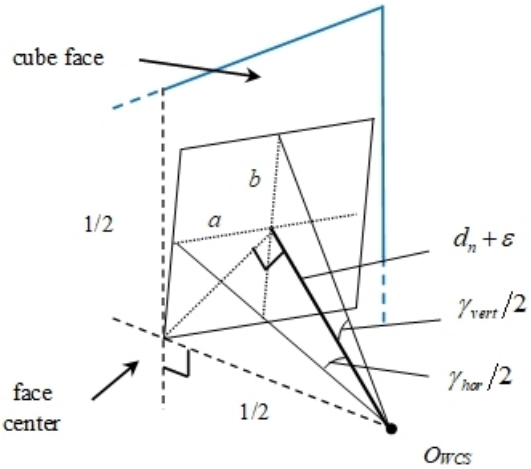


Fig. 3. Determining the distance to the near clipping plane

Then, for the distance d_n the equation can be written:

$$(d_n + \varepsilon)^2 = 0.5^2 - (a^2 + b^2) = \\ = 0.5^2 - (d_n + \varepsilon)^2(\mathrm{tg}^2(\gamma_{hor}/2) + \mathrm{tg}^2(\gamma_{vert}/2)). \quad (2)$$

Substitute Eq. (1) into Eq. (2) and find the distance d_n:

$$d_n = \frac{1}{2\sqrt{1+\mathrm{tg}^2(\gamma_{hor}/2)(1+1/aspect^2)}} - \varepsilon\,. \qquad (3)$$

The stage described is performed when starting 360-video, as well as each time the user changes γ_{hor} or *aspect*.

Table *H* of cube vertices visibility

To simplify checking cube edges visibility, we create a table H that stores boolean flags of being each cube vertex in "+" half-space (where frustum is located) of each clipping plane of camera C_V, except the far plane. Relative to this plane all cube vertices lie obviously in "+" half-space (see d_n determination in Section 3.1). Table H consists of 8 rows (the row index is the cube vertex number), and each row stores 6 flags $b_0,\ldots,b_5$:

- in the subgroup $b_0,\ldots,b_4$, raised b_ith flag means that the vertex lies in the "+" half-space or coincides with the ith clipping plane (0 - near, 1 - left, 2 - right, 3 - bottom, 4 - top);
- flag b_5 is raised if all flags $b_0,\ldots,b_4$ are raised, i.e. the cube vertex is in the frustum of camera C_V.

Consider some cube vertex P. Denote by P_{WCS} its coordinates in WCS, and by $\boldsymbol{p}$, the vector $\boldsymbol{O_{WCS}P_{WCS}}$. The calculation of flags $b_0,\ldots,b_5$ for the vertex P is done by the following

Algorithm A1 to fill a row of the table H

1. Find the projection $p_v = (\boldsymbol{p}, \boldsymbol{v})$ of the vector $\boldsymbol{p}$ on the "view" vector $\boldsymbol{v}$ of camera C_V.
2. Find the projection p_r of the vector $\boldsymbol{p}$ on the "right" vector $\boldsymbol{r}$ similarly to p_v.
3. Find the projection p_{up} of the vector $\boldsymbol{p}$ on the "up" vector $\boldsymbol{u}$ similarly to p_v.
4. Calculate the size d_{hor} of horizontal FOV of camera C_V on the line of the vertex P (see. Fig. 4):
$$d_{hor} = 2p_v\mathrm{tg}(\gamma_{hor}/2)\,.$$
5. Calculate the size d_{vert} of vertical FOV of camera C_V on the line of the vertex P similarly to d_{hor}.
6. $b_0 = (p_v \geq d_n)$, $b_1 = (p_r \geq -d_{hor}/2)$, $b_2 = (p_r \leq d_{hor}/2)$,
$b_3 = (p_{up} \geq -d_{vert}/2)$, $b_4 = (p_{up} \leq d_{vert}/2)$.
7. $b_5 = (b_0 \,\&\&\, b_1 \,\&\&\, b_2 \,\&\&\, b_3 \,\&\&\, b_4)$.

Having executed algorithm *A1* for each cube vertex in order, we obtain table H.

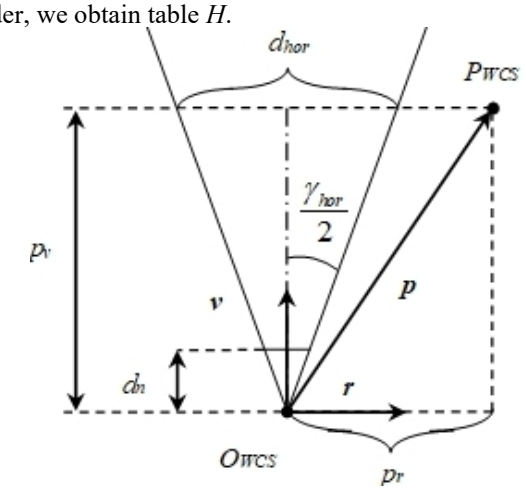


Fig. 4. Checking vertex P visibility

Visible face pairs extraction

Every face pair which common edge intersects the frustum of camera C_V will be extracted for visualization. This is possible in two cases:

(a) if at least one of edge vertices falls into the frustum. This case can be easily detected by checking flags b_5 of edge vertices in the table H (at least one vertex having true b_5 is enough);

(b) if both vertices lie outside the frustum, but the edge intersects at least one clipping plane of camera C_V, and their intersection point falls into the frustum. Divide this case into 3 steps: 1) determining the fact of the intersection of the ith clipping plane by the edge; 2) finding coordinates P_I of their intersection point; 3) checking falling the point P_I into the frustum.

Step 1. The fact of "edge - ith clipping plane" intersection can be easily established using the table H. For this, edge vertices should have opposite flags b_i. Note, if the edge lies in the ith clipping plane, and edge vertices are outside the frustum, then intersection fact with this plane will not be established (both flags b_i will be true), but it will be surely done with other clipping plane, so the case (b) will be correctly detected. Another important thing is that any cube edge isn't able to cross the near or far base of the frustum (see Section 3.1), so there is no need to check these planes.

Step 2. After establishing the fact that cube edge intersects the ith clipping plane, for instance, the right plane (for the remaining planes the derivation will be similar), we need to calculate coordinates P_I of their intersection point. For this, we introduce the following denotations: P_A, P_B - the coordinates of edge vertices in WCS; $\boldsymbol{e}$ - unit vector $\boldsymbol{P_AP_B}$; $\boldsymbol{p_A}$ - the vector $\boldsymbol{O_{WCS}P_A}$; $\boldsymbol{p_I}$ - the vector $\boldsymbol{O_{WCS}P_I}$; $p_{I,r}$ - the projection of the vector $\boldsymbol{p_I}$ on the vector $\boldsymbol{r}$ of camera C_V; $p_{I,v}$ - the projection of the vector $\boldsymbol{p_I}$ on the vector $\boldsymbol{v}$ of camera C_V. In this example, the point P_I lies in the right clipping plane, therefore its projection $p_{I,r}$ is equal to half the size of the horizontal FOV on the line of the point P_I (similarly to size d_{hor} in Fig. 4):

$$p_{I,r} = p_{I,v}\mathrm{tg}\frac{\gamma_{hor}}{2} \quad \text{or} \quad (\boldsymbol{p_I}, \boldsymbol{r}) = (\boldsymbol{p_I}, \boldsymbol{v})\mathrm{tg}\frac{\gamma_{hor}}{2}\,. \qquad (4)$$

Using the distributive property of the dot product of vectors rewrite the Eq. (4) as

$$(\boldsymbol{p_I}, \boldsymbol{\chi_{right}}) = 0\,, \quad \text{where } \boldsymbol{\chi_{right}} = \boldsymbol{r} - \mathrm{tg}(\frac{\gamma_{hor}}{2})\boldsymbol{v}\,. \qquad (5)$$

Write another expression for the vector $\boldsymbol{p_I}$ using the vector parametric equation of the line P_AP_B:

$$\boldsymbol{p_I} = \boldsymbol{p_A} + t_I\boldsymbol{e}\,, \qquad (6)$$

where t_I is the parameter determining the position of the point P_I on the line P_AP_B. Substitute Eq. (6) into Eq. (5) and find t_I:

$$t_I = -(\boldsymbol{p_A}, \boldsymbol{\chi_{right}}) / (\boldsymbol{e}, \boldsymbol{\chi_{right}})\,. \qquad (7)$$

Similarly to Eq. (6) write for coordinates P_I the expression $P_I = P_A + t_I\boldsymbol{e}$. Substitute t_I from Eq. (7) in it and find required coordinates P_I:

$$P_I = P_A - \frac{(\boldsymbol{p_A}, \boldsymbol{\chi_{right}})}{(\boldsymbol{e}, \boldsymbol{\chi_{right}})}\boldsymbol{e}\,. \qquad (8)$$

As one can notice, the coordinates of intersection points of the edge P_AP_B with the left, top and bottom

clipping planes will differ from Eq. (8) only by similar terms χ_{left}, χ_{top} and χ_{btm}. The expressions of these terms are derived in a similar way:

$$\chi_{left} = \boldsymbol{r} + \mathrm{tg}(\frac{\gamma_{hor}}{2})\boldsymbol{v}, \quad \chi_{top} = \boldsymbol{u} - \mathrm{tg}(\frac{\gamma_{vert}}{2})\boldsymbol{v}, \quad \chi_{btm} = \boldsymbol{u} + \mathrm{tg}(\frac{\gamma_{vert}}{2})\boldsymbol{v}. \tag{9}$$

Step 3. Having the coordinates P_I of intersection point, we check falling the point P_I into the frustum of camera C_V. To do this, we calculate the flag b_5 for the point P_I using algorithm *A1*, and check its value (*true* means visible edge). Note, when calculating flag b_5, the calculation of the flag b_i can be omitted, as the point P_I lies in the ith plane, and b_i will obviously be *true*.

Based on the cases (a) and (b) considered, we define the algorithm to check the visibility of the edge {*m*, *n*}, where *m*, *n* are the numbers of edge vertices, introduced at the beginning of the Section 3. Denote by b_{edge} the flag of edge {*m*, *n*} visibility. The calculation of b_{edge} is done by the following

Algorithm A2 to determine the edge {m, n} visibility

1. Check the visibility of edge vertices (using the table *H*):
   If ($H_{m,5}$ || $H_{n,5}$) is *true*, then:
   b_{edge} = *true*, exit the algorithm.
2. Check, whether the edge lies in the "-" half-space of any clipping plane:
   Loop by *i* from 0 to 4, where *i* is plane number
   If (!$H_{m,i}$ && !$H_{n,i}$) is *true*, then:
   b_{edge} = *false*, exit the algorithm.
   End Loop.
3. Check the presence of at least one visible point P_I of the intersection of the edge with any clipping plane
   Loop by *i* from 1 to 4
   If ($H_{m,i}$ ^ $H_{n,i}$) is *true*, then:
   Calculate coordinates P_I by Eq. (8) and (9).
   Calculate the flag b_5 by algorithm *A1*.
   If b_5 is *true*, then:
   b_{edge} = *true*, exit the algorithm.
   End If.
   End Loop.
4. b_{edge} = *false*.

Next, using algorithm *A2*, visible face pairs are extracted. Denote by B_{faces} the boolean array of 6 flags of cube faces visibility (true/false - the face is visible/not visible), by *D* and *E* - the arrays of face pairs and cube edges, introduced at the beginning of the Section 3, and by b_{pair} - the flag of at least one visible face pair. Execute the following

Algorithm A3 to extract visible face pairs

1. Clear array B_{faces} with value *false*, b_{pair} = *false*.
2. Loop by *j* from 0 to 11, where *j* is the edge index
   Calculate flag b_{edge} of the edge *E*[*j*] by algorithm *A2*.
   If b_{edge} is *true*, then:
   B_{faces}[*D*[*j*][*0*]] = *true*.
   B_{faces}[*D*[*j*][*1*]] = *true*.
   b_{pair} = *true*.
   End If.
   End Loop.

If algorithm *A3* results in *true* flag b_{pair}, then we proceed to the stage of visualization of the faces marked in B_{faces} (see the Section 3.5). If b_{pair} is *false* (no visible face pairs were found), this means that camera C_V captures some single cube face and we need to extract it for visualization (see the Section 3.4).

Extraction of single visible face

As one can see, the visible one will be cube face with the smallest angle between the external normal and the vector $\boldsymbol{v}$ of camera C_V. To determine the number of such a face, we calculate the cosines of the angles between the normals to the faces and the vector $\boldsymbol{v}$, and extract the face with the largest cosine. Denote by *K* the array of cosines for faces 0-5, and by $\boldsymbol{n}_2$, $\boldsymbol{n}_3$ and $\boldsymbol{n}_5$ the normals to the back, right, and top cube faces. Write the sequence of normals for the faces 0-5: {$-\boldsymbol{n}_5$, $-\boldsymbol{n}_2$, $\boldsymbol{n}_2$, $\boldsymbol{n}_3$, $-\boldsymbol{n}_3$, $\boldsymbol{n}_5$}. Since the normals $\boldsymbol{n}_2$, $\boldsymbol{n}_3$, $\boldsymbol{n}_5$ coincide with the axes OZ_{WCS}, OX_{WCS}, OY_{WCS}, the calculation of the array *K* reduces to writing the sequence of the vector $\boldsymbol{v}$ coordinates with signs from the normals' sequence. Execute the following

Algorithm A4 to extract single visible kth face

1. *K* = {$-\boldsymbol{v}_{obs,y}$, $-\boldsymbol{v}_{obs,z}$, $\boldsymbol{v}_{obs,z}$, $\boldsymbol{v}_{obs,x}$, $-\boldsymbol{v}_{obs,x}$, $\boldsymbol{v}_{obs,y}$}.
2. *k* = 0. // by default 0th face is supposed visible.
3. Loop by *i* from 1 to 5, where *i* is face number (see Fig. 1).
   If *K*[*i*] > *K*[*k*], then *k* = *i*.
   End Loop.
4. B_{faces}[*k*] = *true*.

After executing the algorithm *A4*, array B_{faces} will contain one true flag marking single visible cube face. Visualization of the faces marked in B_{faces} is performed at the next stage.

Visualization of extracted faces

To each element of the array B_{faces} the face texture of *d* x *d* pixels is corresponded, and all 6 face textures, as noted in the Section 2, are merged into 360-frame of 3*d* x 2*d* pixels. At this stage, face textures marked in B_{faces} will be extracted from 360-frame and applied to cube model. Face textures will be extracted to an array *T* of 6 texture objects (one object per cube face). Each element of the array *T* is a continuous area of VRAM, allocated for storing one face texture. It is important to note here that in 360-frame each face texture is stored not in one continuous line, but in a number of *d* substrings of *d* pixels in length (see Fig. 5). Since transferring a large number of small data pieces into VRAM reduces the GPU's performance, we tune video driver (using operator *glPixelStorei* of the OpenGL library) so that the substrings of face texture are automatically merged and transferred to VRAM as one continuous piece. This is done in the following

Algorithm A5 to visualize 360-frame

1. Clear frame buffer, set the viewport, as well as projection and modelview matrices according to camera C_V's params.
2. Loop by *i* from 0 to 5, where *i* is face number.
   If B_{faces}[*i*] is *true*, then:

Set row length n_{RL} of 360-frame, the number n_{SR} of skipped rows and the number n_{SP} of skipped pixels in a row (see Fig. 5):

glPixelStorei (…*_ROW_LENGTH*, 3*d)*,
glPixelStorei (…*_SKIP_ROWS*, $\lfloor i/3 \rfloor d$),
glPixelStorei (…*_SKIP_PIXELS*, (*i* % 3)*d*),

where "…" is shortening of GL_UNPACK.

Load *i*th face texture to *T*[*i*]th texture object by means of the operator *glTexSubImage2D*.

Render *T*[*i*]th texture object on the *i*th face.

End If.

End Loop.

Note, if 360-frame isn't changed during the visualization process (for example, the video is paused), then in step 2 of the algorithm *A5*, the same face textures are not repeatedly loaded into VRAM, but the previously loaded ones are used.

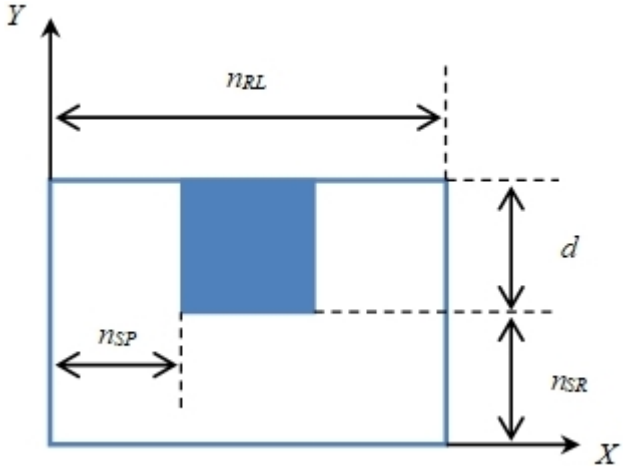
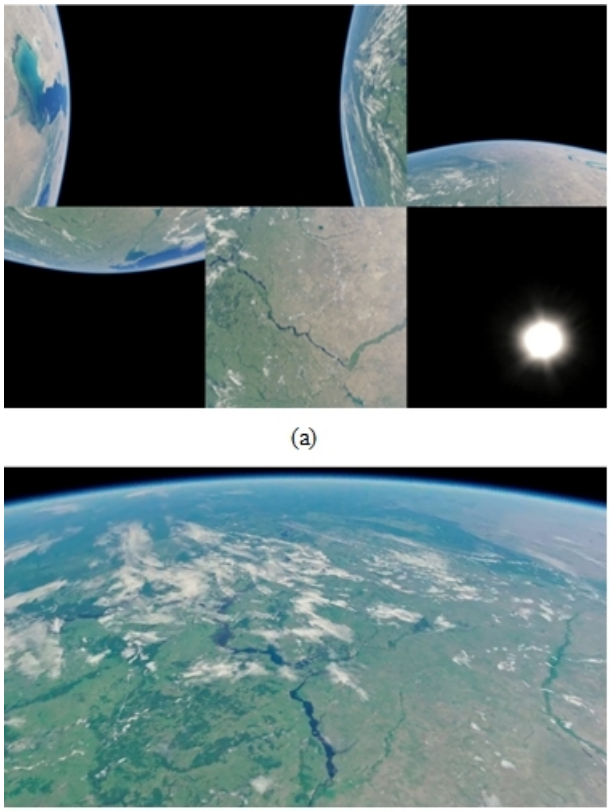


Fig. 5. Reading face texture from 360-frame

4. Results

The proposed technology and algorithms were implemented in a software complex (360-player) written in C++ language using the OpenGL graphics library. The player performs high-quality visualization of a virtual environment from a 360-video based on cubemap projection. During the visualization, the viewer can rotate the camera corresponding to tilting the head up/down and left/right, as well as change camera FOV (viewing angle and aspect).

The developed solution was tested on 360-video with a resolution of 3000x2000 pixels, created in the visualization system for virtual experiments to observe the Earth from the International Space Station (ISS) [3]. By means of the player developed, an experiment was reproduced where the researcher, rotating the observation tool, searches and analyzes a number of Earth objects along the ISS daily track. Fig. 6a shows an example of 360-video frame, and Fig. 6b shows visualization of this frame in 360-player.



(a)

(b)

Fig. 6. Testing the solution developed: (a) a frame of source 360-video; (b) VE visualized from the frame

5. Conclusions

The paper considers the task of increasing the efficiency of VE visualization using 360-degree video based on cubemap projection. High-quality visualization (providing the effect of immersion into VE) requires snapshots of high-resolution VE cubemap, which causes overloading graphics card and impedes smooth changing the frames. To solve this task, an effective technology is proposed, based on the extraction and visualization of visible cube faces, which can significantly reduce the amount of data sent to graphics card without any loss of visual quality. The paper proposes algorithms to extract visible cube faces both in the case of falling cube edges into FOV, and in the case of the absence of visible edges (the case of single visible face). The resulting technology and algorithms were implemented in a software and tested on 360-video containing the visualization of virtual experiment on observing the Earth from space. The testing of the software confirmed the correctness of the solution obtained, as well as its applicability for virtual environment systems and scientific visualization, video simulators, virtual laboratories, etc. In the future, we plan to expand the results to increase the efficiency of visualization of VE projected onto the dodecahedron.

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] Bondarev A.E., Galaktionov V.A. Construction of a Generalized Computational Experiment and Visual Analysis of Multidimensional Data // CEUR Workshop Proceedings: Proc. 29th Int. Conf. Computer Graphics and Vision (GraphiCon 2019), Bryansk, 2019, vol. 2485, p. 117-121., http://ceur-ws.org/Vol-2485/paper27.pdf.

[2] Gavrilov N., Turlapov V. General implementation aspects of the GPU-based volume rendering algorithm // Scientific Visualization. - 2011. - Vol. 3, № 1. - p. 19-31.

[3] Mikhaylyuk, M.V., Timokhin, P.Y., Maltsev, A.V. A method of Earth terrain tessellation on the GPU for space simulators // Programming and Computer Software - 2017. - Vol. 43, p. 243-249. DOI: 10.1134/S0361768817040065.

[4] Mikhaylyuk M.V., Timokhin P.Yu. Memory-effective methods and algorithms of shader visualization of digital core material model // Scientific Visualization - 2019. - Vol. 11, № 5. - p. 1-11. DOI: 10.26583/sv.11.5.01.

[5] Porter M. Galactic Center Visualization Delivers Star Power // https://chandra.harvard.edu/photo/2019/gcenter/ (review date 25.05.2020).

[6] El-Ganainy T., Hefeeda M. Streaming Virtual Reality Content // https://www.researchgate.net/publication/3119

[7] 25694_Streaming_Virtual_Reality_Content (review date 25.05.2020).

[8] Kuzyakov E., Pio D. Next-generation video encoding techniques for 360 video and VR // https://code.facebook.com /posts/1126354007399553/next-generation-video-encodin (review date 25.05.2020).

About the authors

Timokhin Petr Yu., senior researcher of Federal State Institution «Scientific Research Institute for System Analysis of the Russian Academy of Sciences». E-mail: webpismo@yahoo.de.

Mikhaylyuk Mikhail V., Dr. Sc. (Phys.-Math.), chief researcher of Federal State Institution «Scientific Research Institute for System Analysis of the Russian Academy of Sciences». E-mail: mix@niisi.ras.ru.