

Visualization of Hybrid Virtual Scenes Using Hardware-Accelerated Ray Tracing and Rasterization

P.Yu. Timokhin^{1,A}, M.V. Mikhaylyuk^{2,A}

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

¹ ORCID: 0000-0002-0718-1436, p_tim@bk.ru

² ORCID: 0000-0002-7793-080X, mix@niisi.ras.ru

Abstract

This paper discusses the task of real-time visualization of hybrid virtual scenes that combine traditional polygonal scenes and procedural objects. An approach to solve this task on modern multicore graphics processors is proposed, in which the visualization of the polygonal scene is performed via rasterization and non-polygonal procedural objects - using hardware-accelerated ray tracing. The paper describes developed methods and algorithms allowing two such synthesized images to be merged by depth using the bundle of programming interfaces Vulkan-OpenGL. Based on the proposed methods and algorithms, software complex prototype, implementing real-time visualization of hybrid scenes, was created. The complex was tested on a number of scenes with complicated procedural objects specified by 3D scalar fields (point clouds). The results of the approbation confirmed the feasibility of obtained solutions and their applicability to virtual environment systems, scientific visualization, virtual laboratories, video simulators, educational applications, etc.

Keywords: virtual environment system, real-time, hybrid scene, ray tracing, depth buffer, GPU.

1. Introduction

One of the most important tasks of modern virtual environment systems [1-3] is visualization of *virtual prototypes* of complex real objects and phenomena in real-time (at a speed of at least 25 frames per second). Traditionally, such visualization is performed based on the rasterization of *polygonal 3D scenes* in which virtual objects are assembled from interconnected triangular graphic primitives (polygons). This approach is well suited for virtual prototyping of complex technical systems, architectural facilities, human avatars and many other objects [4-6], and, besides that, polygon rasterization has hardware support in modern multicore graphics processors (GPUs). However, under conditions of constantly growing demands on virtual environment quality, using polygonal assemblies as the only ones is not sufficient. There are a number of important objects of the virtual environment for which the pre-calculated triangulation is either not suitable by definition, or is impractical due to the critical growth of the polygonal complexity of the scene, that violates real-time visualization (examples of such objects are volumetric clouds, water surfaces, forests, grass, etc.).

An effective way to solve this problem is the construction of so-called *hybrid* (heterogeneous [7]) virtual scenes in which traditional polygonal scenes are combined with *procedural objects* calculated during visualization. In virtual environment systems, such objects are often formed using dynamic polygonal techniques based on GPU-rasterization (adaptive triangulation [8], particle systems [9], vertex animation [10], etc.). With the invention of *hardware-accelerated ray tracing* [11], high-quality *non-polygonal* modeling and visualization of complex procedural objects became possible to be performed in real-time on the GPU (for in-

stance, objects specified by height maps [12] or 3D scalar fields (point clouds) [13]). This technology, together with GPU-rasterization, forms a powerful computing potential for visualization of hybrid virtual scenes, however, in order to use it, two fundamentally different image synthesis methods - rasterization and ray tracing, are necessary to be combined. This paper proposes approach, methods and algorithms that allow a frame of visualization of a scene comprising non-polygonal procedural objects, synthesized using hardware-accelerated ray tracing, and a frame of visualization of a polygonal scene, obtained by means of rasterization, to be merged by depth.

2. Hardware and software background

In this research, the combination of rasterization and hardware-accelerated ray tracing is implemented based on a pair of programming interfaces (APIs) - OpenGL and Vulkan.

The OpenGL standard [14] has an architecture approved over the years and an intuitive interface, provides the ability to parallelize computations on the GPU using the GLSL (OpenGL Shading Language) [15], supports cross-platform and is distributed under a free license. All this makes OpenGL a reliable, convenient and in-demand 3D graphics programming tool, despite "freezing" phase of its active development in 2017 [16]. The heart of OpenGL-visualization is a programmable graphics pipeline (hereinafter *GL-pipeline*) based on rasterization, which runs in parallel on thousands of unified cores (CUDA-cores) of the GPU.

The Vulkan standard [17] is being developed by the same Khronos Group industrial consortium that supervises OpenGL, however, is based on a fundamentally different architecture, more flexible and low-level. Thanks to the deeply tunable architecture, Vulkan provides low-overhead graphics computing, as well as more complete control over the GPU and less CPU usage. On the one hand, Vulkan's low-level ideology makes the graphics programming process laborious [18] and requires in-depth educating of specialists [19], and, on the other hand, opens up the possibility of integrating innovative graphics technologies into the standard that go beyond the existing OpenGL paradigm. This applies, in particular, to the hardware-accelerated ray tracing used in this work (also referred to as the *RTX architecture*).

The hardware-software RTX architecture is implemented in NVidia GPUs (starting from the Turing generation [20]) and is based on parallelization of ray tracing [21, 22] on RT-cores - GPU computing cores of a new type. Unlike unified CUDA-cores designed for a wide range of graphical and non-graphical computations, RT-cores are focused on efficient performing a narrow range of tasks related only to ray tracing. In modern GPUs, the number of RT-cores is as yet significantly lower than the number of CUDA-cores: the flagship GPU GeForce RTX 4090 contains 16384 CUDA-cores and only 128 RT-cores. Nevertheless, with each subsequent GPU generation, the number and efficiency of RT-cores increases several times [23]. RT-cores are accessed through a special ray tracing pipeline (hereinafter *RT-pipeline*), which includes closed (hardware) and open (programmable) stages (for more details, see [24]). Currently, the RT-pipeline can be programmed using APIs NVidia OptiX, Microsoft DXR or Vulkan [25], among which only Vulkan is open-source and allows the functionality of hardware-accelerated ray tracing to be fully revealed [26].

3. Merged by depth visualization of hybrid scenes

3.1. Proposed approach

Suppose there is a hybrid scene represented as a combination of two virtual scenes: a) comprising polygonal objects (both pre-assembled and procedural), and b) a scene complementing first one, which comprises non-polygonal procedural objects. Herewith, rendering of the first scene is performed on the GL-pipeline (hereinafter called *GL-scene*), and rendering of the second scene is performed on the RT-pipeline (hereinafter called *RT-scene*).

We will consider the task of merging GL- and RT-scene renderings by depth within the Vulkan-OpenGL hybrid visualization system we developed earlier [27]. This system outputs

Vulkan-visualization into an OpenGL-window based on interoperability mechanism (ability to interact) of APIs Vulkan and OpenGL, developed by the Khronos Group consortium [28]. According to this mechanism, synthesized images are transferred from Vulkan to OpenGL via *shared video memory area*, and Vulkan and OpenGL access to shared resources (GPU and video memory) is synchronized using a pair of *shared semaphores*. To implement a depth-merged rendering of GL- and RT-scene in such a system, it is necessary to overcome a number of obstacles. Firstly, the RT-scene rendering is performed without a depth buffer, i.e. the RT-scene visualization frame (RT-frame) does not contain depth data (due to in the RT-pipeline, distances to objects are calculated and compared "on the fly" in parametric form). Secondly, the result of the RT-scene rendering (ray colors) is written in a 4-channel format (rgba), which does not involve storing auxiliary depth data (see the format of the *imageStore* function built in the GLSL language [15]). Thirdly, according to the above-mentioned interoperability mechanism, the RT-frame is imported from the shared video memory area into OpenGL in the form of 2D texture (see the *glTextureStorageMem2DTEXT* function [29]), which also does not involve storing auxiliary depth data. In this regard, the task arises as to construct an RT-frame with RT-scene depth and rgba-color data, to pack it into a 4-channel 2D texture, as well as to render such a packed RT-frame into OpenGL framebuffer.

To solve this task, in this work three new blocks are proposed to be introduced into the hybrid visualization system [27]: **block D** for synthesizing RT-scene depth data, **block P** for packing RT-scene depth and rgba-color data into the RT-frame, and **block U** for rendering packed RT-frame. Let's consider operation scheme of such an extended system (see Figure 1).

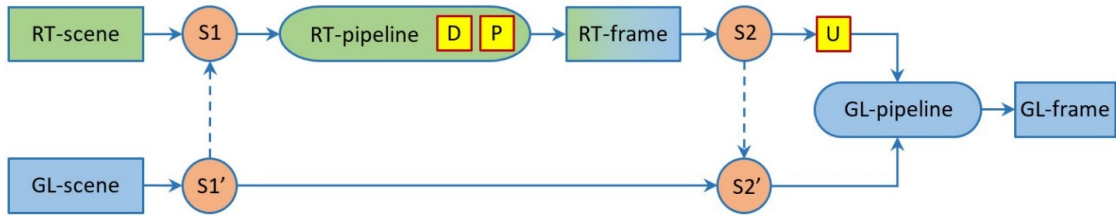


Fig. 1. The operation scheme of the extended hybrid visualization system.

In the Figure 1, S1 and S2 denote the shared semaphores used on the Vulkan side, and S1' and S2' denote their doubler semaphores on the OpenGL side. Initially, all semaphores are in a standby state blocking rendering of the GL- and RT-scene.

When starting the visualization stage, the S1' doubler semaphore on the OpenGL side is switched to a signal state, meaning that synthesis of the previous GL-frame has been completed on the GPU (or visualization system initialization has been completed, if this is the first launch). Activation of the S1' doubler semaphore automatically removes the lock from the S1 semaphore on the Vulkan side and starts rendering the RT-scene on the RT-pipeline (the final stage of this process is the execution of blocks D and P). Until the RT-pipeline finishes its work (i.e. the packed RT-frame will not be fully formed), the S2 semaphore is in the standby state and blocks GL-scene rendering.

After activating the S2 semaphore, the lock is automatically removed from the S2' doubler semaphore on the OpenGL side and the process of rendering the packaged RT-frame and GL-scene on the GL-pipeline is started. The visualization of the packed RT-frame is implemented by means of the U block and includes unpacking the RT-frame, superimposing it as a 2D texture on a polygonal model of the screen-sized rectangle and rendering the resulting model using the GL-pipeline to OpenGL framebuffer. Note that after activation, each of the shared semaphores is automatically reset to the standby state, which allows it to be used at the next visualization frame.

The blocks D, P and U introduced above are implemented based on the developed methods of: a) synthesis of RT-scene depth data, b) packing RT-scene rgba-color and depth data, and c) rendering packed RT-frame. Let's consider these methods.

3.2. Method to synthesize RT-scene depth data

As noted above, when synthesizing an image by means of hardware-accelerated ray tracing, the mutual occlusion of virtual objects is computed inside the RT-pipeline based on a parametric form:

$$P_{hit} = P_{eye} + r t_{hit}, \quad (1)$$

where P_{eye} is ray origin (coincides with virtual camera position), r is ray direction (passes through a certain point of shaded pixel, for example, its center), P_{hit} is the point of the intersection of the ray with virtual object, and t_{hit} is parameter. During tracing¹, the RT-pipeline extracts the point closest to the observer, i.e. with the lowest t_{hit} parameter value (hereinafter referred to as the P_{chit} point with the t_{chit} parameter), and shades the pixel with the color of this point or with "miss" color, if there are no objects along ray path (case $t_{chit} \leq 0$).

To synthesize RT-scene depth data in the format used in OpenGL depth test and depth buffer [30], the t_{chit} parameter should be transformed into the $z_w \in [0,1]$ coordinate of the P_{chit} point in OpenGL window coordinate system. This is done step-by-step, by passing through the chain of OpenGL coordinate systems [31].

First, we transform the t_{chit} parameter to the z_{vcs} coordinate of the P_{chit} point in **view coordinate system** (VCS system). To do this, we express the z_{vcs} coordinate from Eq. (1), taking into account that in the VCS system the P_{eye} point has coordinates (0,0,0):

$$z_{vcs} = r_z t_{chit}, \quad (2)$$

where the unknown is the r_z coordinate of the r ray direction vector in the VCS system.

To find the r_z value, we establish the relationship of the r vector coordinates in the VCS system with the normalized screen coordinates $u, v \in [-1,1]$ of the center of the shaded pixel (see Figure 2). The (u, v) coordinates are set in the *UV system*, which origin coincides with viewport center, the U axis coincides in the direction with the camera's c_r "right" ort, and the V axis is directed opposite to the camera's c_{up} "up" ort:

$$(u, v) = (((2j+1)/w) - 1, ((2i+1)/h) - 1), \quad (3)$$

where i and j are row and column indices of the shaded pixel ($i=0$ and $j=0$ correspond to the upper-left pixel), and w and h are viewport width and height, in pixels.

Due to picture plane in which the (u, v) point lies is drawn through the point at the end of the c_v ort of the camera's "view" direction, the vector r can be expressed as

$$r = \|c_v + d_w u c_r - d_h v c_{up}\|, \quad (4)$$

where $d_h = \tan(0.5\varphi)$, $d_w = (w/h)d_h$, and φ is vertical field of view angle (known initially). Substituting in (4) the coordinates of the vectors $c_v = (0,0,-1)$, $c_r = (1,0,0)$, $c_{up} = (0,1,0)$ in the VCS system, we obtain an expression for the coordinates of the r vector:

$$(r_x, r_y, r_z) = \text{normalize}(d_w u, -d_h v, -1), \quad (5)$$

where *normalize* is an efficient vector normalization function built in the GLSL language [15].

¹ In this work, ray tracing is performed in an opaque RT-scene.

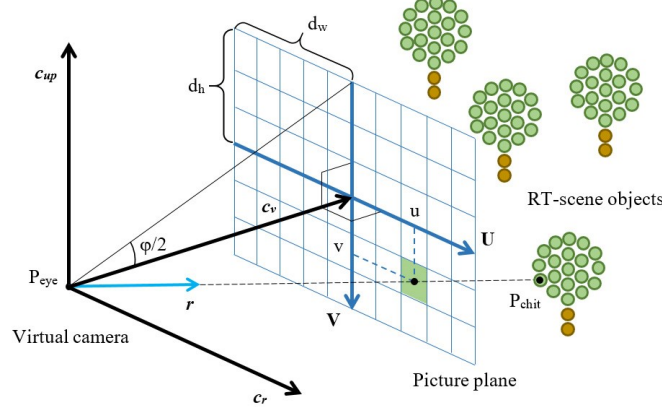


Fig. 2. Normalized screen coordinates (u, v) of the center of the pixel shaded by ray tracing.

Next, according to the definition of the OpenGL perspective projection matrix [32], we find transformations of the z_{vcs} coordinate obtained using (2) and (5) into z_{ccs} and w_{ccs} coordinates of the P_{chit} point in **clip space coordinate system** (CCS system):

$$z_{ccs} = -\frac{f_{vcs} + n_{vcs}}{f_{vcs} - n_{vcs}} z_{vcs} - 2 \frac{f_{vcs} n_{vcs}}{f_{vcs} - n_{vcs}}, w_{ccs} = -z_{vcs}, \quad (6)$$

where n_{vcs} , f_{vcs} are distances to near and far clipping planes in the VCS system (known initially).

The values z_{ccs} and w_{ccs} are related to the $z_{ndcs} \in [-1, 1]$ coordinate of the P_{chit} point in **normalized device coordinate system** (NDCS system) [31]:

$$z_{ndcs} = z_{ccs} / w_{ccs}, \quad (7)$$

which, in turn, is directly related to the desired z_w value:

$$z_w = 0.5(f_w - n_w)z_{ndcs} + 0.5(f_w + n_w), \quad (8)$$

where $n_w = 0$, $f_w = 1$ are distances to near and far clipping planes in window coordinate system.

Substituting expressions (7), (6), (2) and the r_z value calculated using (5) in expression (8), we obtain the desired expression of the z_w coordinate through the t_{chit} parameter:

$$z_w = \frac{1}{f_{vcs} - n_{vcs}} \left(\frac{f_{vcs} n_{vcs}}{r_z t_{chit}} + 0.5(f_{vcs} + n_{vcs}) \right) + 0.5.$$

The calculation of the z_w coordinate is implemented at the Ray Generation Shader stage (RG- shader) of the RT-pipeline (see [24]) after executing the built-in ray tracing function *traceRayEXT* and before writing tracing result to the image.

3.3. Method to pack RT-scene rgba-color and depth data

After completing the synthesis of RT-scene depth data, 5 data channels are formed: ray color components r , g , b , a (the alpha channel is used to mask RT-scene objects) and the depth z_w . To pack this data into the RT-frame (see Section 3.1), we use a 4-channel 2D texture, each channel of which has the format of a 16-bit unsigned integer (in Vulkan this format is referred to as `VK_FORMAT_R16G16B16A16_UINT`, and in OpenGL - `GL_RGBA16UI`).

Packing the r, g, b, a, z_w values in a texel of such format is implemented based on developed diagram shown in Figure 3.

In the diagram, the input r, g, b, a, z_w values are 32-bit normalized real numbers (float type), and the output r', g', b', a' values are 16-bit unsigned integers (in Fig. 3, one cell corresponds to 8 bits). Packing is done at the RG-shader stage with the help of a number of built-in GLSL-functions [15]:

1. Using the *packUnorm4x8* function, convert each of the components r, g, b, a to an 8-bit normalized unsigned integer type (used at automatic color compression in OpenGL [33]) and merge the resulting numbers into a 32-bit unsigned integer A .

2. Using the *floatBitsToUint* function, convert the real depth z_w to a 32-bit unsigned integer B (while bit representation of the real number remains unchanged).

3. Transfer the lower and upper 16 bits of the integer A to the components r' and g' , and the lower and upper 16 bits of the integer B - to the components b' and a' :

$$(r', g', b', a') = (A \& 0xFFFF, (A \gg 16) \& 0xFFFF, B \& 0xFFFF, (B \gg 16) \& 0xFFFF),$$

where "&" is a bitwise "AND" operation, and ">>" is a bitwise right shift operation.

4. Using the *imageStore* function, write the r', g', b', a' values into the 2D texture of the RT-frame.

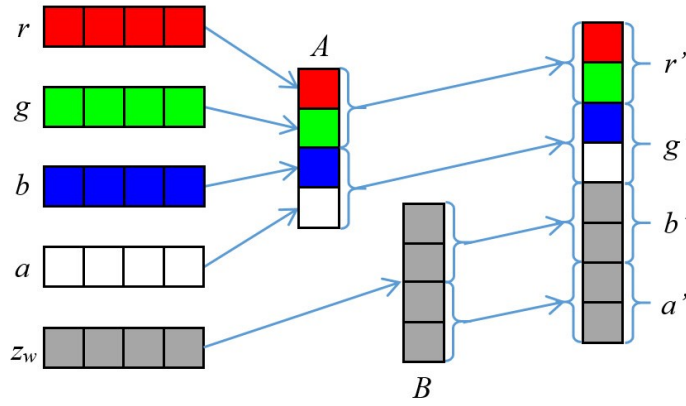


Fig. 3. The diagram of packing rgba-color and depth z_w into a texel of a 4-channel 2D texture.

After applying the described algorithm to all viewport pixels, a packed RT-frame containing OpenGL-formatted rgba-color and depth data of the RT-scene is formed in the shared video memory area (see Section 3.1).

3.4. Method to render the packed RT-frame

To visualize the packed RT-frame, in this work a polygonal rectangle model of viewport size, as well as a developed fragment shader program (*F-shader*), are used. Using hardware rasterization on the GL-pipeline, the rectangle model is transformed into image fragments, for each the F-shader calculates color and depth based on data packed in the 2D texture of the RT-frame. The F-shader algorithm includes unpacking these data (using built-in functions of the GLSL language [15], as in Section 3.3) and is represented as follows:

1. Form the texture coordinates (s', t') of the fragment in Vulkan texture coordinate system (the 2D texture of the RT-frame is synthesized in this system):

$$(s', t') = (s, 1 - t),$$

where (s, t) are texture coordinates of the fragment in OpenGL texture coordinate system, which are calculated automatically by the GL-pipeline.

2. Using the *texture* function and texture coordinates (s', t') , obtain encoded integer components r', g', b', a' from the 2D texture of the RT-frame. Despite the texture channels are 16-bit, the *texture* function returns 32-bit integers r', g', b', a' , in which only the lower 16 bits are informative.

3. Restore two 32-bit integers A and B (see Section 3.3) from the r', g', b', a' components:

$$A = (g' \ll 16) | (r' \& 0xFFFF), B = (a' \ll 16) | (b' \& 0xFFFF),$$

where " \ll " is a bitwise left shift operation, and "|" is a bitwise "OR" operation.

4. Using the *unpackUnorm4x8* function, convert the integer A into four (r_n, g_n, b_n, a_n) real normalized color components.

5. Using the *uintBitsToFloat* function, represent the integer B as a real z_w depth.

6. Write the values r_n, g_n, b_n, a_n to the fragment's output color attribute, and the z_w value to the fragment's built-in output depth attribute *gl_FragDepth*.

After the fragment has been processed by the F-shader, it passes through alpha and depth test stages prior to reach OpenGL framebuffer. These stages are enabled in advance and configured² so that alpha test discards masked fragments (with zeroed a_n), and depth test passes into the framebuffer only those fragments whose depths are less than ones previously written in it. After passing through the described stages and rendering of the RT-frame, the GL-scene is rendered (also with depth test enabled), which results in a merged by depth image of the GL- and RT-scene, formed in the framebuffer (see the example in Figure 4).

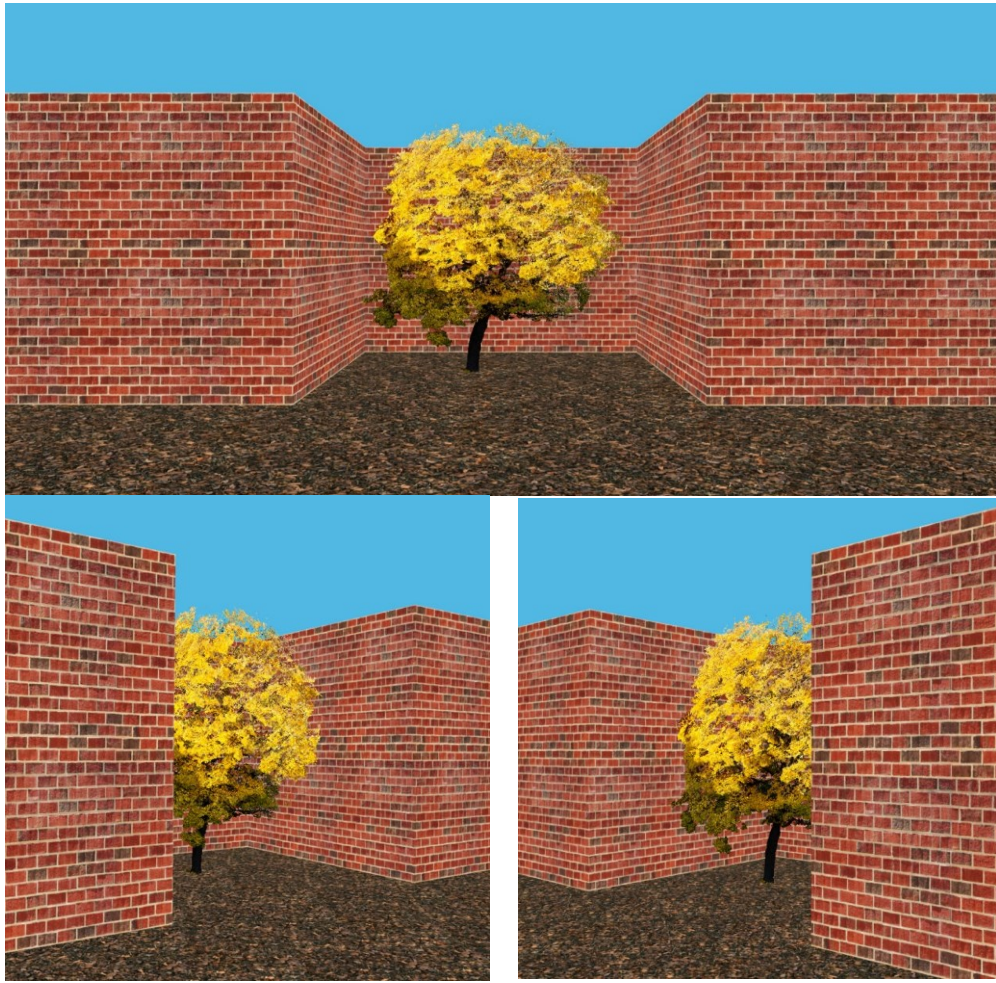


Fig. 4. An example of visualization of a hybrid virtual scene, performed by the developed solution (the tree model is rendered using hardware-accelerated ray tracing, and the models of the wall and the underlying surface - by means of rasterization).

4. Results

Based on the proposed methods, algorithms and approach, a prototype software complex performing real-time visualization of hybrid virtual scenes, was created. The development of this complex was carried out in C++ and GLSL languages using an interoperable bundle of

² To enable alpha and depth test stages *glEnable(GL_ALPHA_TEST)* and *glEnable(GL_DEPTH_TEST)* operators are used, and for configuring ones - *glAlphaFunc(GL_GREATER, 0.1f)* and *glDepthFunc(GL_LESS)*.

graphics APIs OpenGL 4.5 and Vulkan 1.3.241. The created complex includes a basic GL-scene visualizer and a VK-capsule [27] connected to it, which implements the visualization of an RT-scene specified by means of a 3D scalar field (point cloud) [13]. The complex was tested on complicated hybrid scenes which RT-scenes contain millions of points (each point is modeled by a procedural sphere). Figure 4 shows an example of visualization of such a hybrid scene, performed from different angles using the developed complex. The depicted scene contains tree model³ comprising 1580297 points (RT-scene), as well as polygonal models of a wall and underlying surface (GL-scene). Figure 4 shows that these models correctly occlude each other by depth at all angles. The rendering speed of this hybrid scene was about 900 frames per second (the average frame synthesis time is 1.11 milliseconds) at Full HD viewport resolution. Visualization was performed on a personal computer (Intel Core i7-6800K 6x3.40 GHz, 16 GB RAM) equipped with NVidia RTX 2080 graphics card (2944 CUDA-cores, 46 RT-cores, 8 GB VRAM, NVidia DCH 536.40 driver).

5. Conclusions

In modern virtual environment systems, hybrid scenes are actively used, which combine traditional polygonal objects with procedural objects computed during the visualization process. Thanks to the rapid development of the GPU architecture, it has become possible to synthesize high-quality images of such objects in real-time using hardware-accelerated ray tracing. In order to integrate such images (RT-frames) into images generated by virtual environment systems, it is necessary to overcome the fundamental difference between ray tracing and rasterization of polygonal scenes, which remains widespread due to powerful hardware support in the GPU.

To solve this task, an effective approach has been proposed, allowing to merge by depth the RT-frame and visualization frame of the polygonal scene, which is based on an extension of image exchange mechanism between the Vulkan API (ray tracing side) and the OpenGL API (rasterization side). The key advantage of this approach is the ability to isolate ray tracing in a separate plug-in module, which allows virtual environment system to be flexibly configured for a specific task, as well as the hybrid scene visualization system to be effectively developed and maintained. As part of the proposed approach, methods and algorithms have been developed to synthesize the missing depth data of the RT-scene (according to which the RT-frame is rendered), to pack this data together with the color into the RT-frame, as well as to render the packed RT-frame into OpenGL framebuffer. In all the created methods and algorithms, the calculations are parallelized on the GPU, so that merging of the frames is carried out with minimal time.

The results of testing the prototype of the hybrid scene visualizer, implemented based on the created methods and algorithms, confirmed the adequacy of the proposed approach to the task, as well as its applicability to virtual environment systems, scientific visualization, video simulators, geoinformation systems and other computer graphics applications.

6. Acknowledgements

The publication is made within the state task of Federal State Institution “Scientific Research Institute for System Analysis of the Russian Academy of Sciences” on topic No. FNEF-2024-0002 “Mathematical modeling of multiscale dynamic processes and virtual environment systems”.

References

1. UNIGINE 2. Real-time 3D visualization SDK for simulation & training, 2005-2024. (<https://unigine.com/products/sim/advantages/>)

³ The model "Beautiful Autumn" by Epic_Tree_Store [34], distributed under Creative Commons License Attribution 4.0 International (CC BY 4.0) (<http://creativecommons.org/licenses/by/4.0/>).

2. Mikhaylyuk M.V., Maltsev A.V., Timokhin P.Y., Strashnov E.V., Kryuchkov B.I., Usov V.M. The VirSim Virtual Environment System for the Simulation Complexes of Cosmonaut Training // Scientific Journal Manned Spaceflight. – 2020. – No 4(37). – pp. 72–95 [in Russian] (doi: 10.34131/MSF.20.4.72-95)
3. Andreev S.V., Bondarev A.E., Bondareva N.A., Galaktionov V.A., Rykunov S.D., Ustinin M.N. On the Application of Stereoscopic Technologies in Biological Research // Scientific Visualization. – 2023. – Vol. 15, No. 4. – pp. 68–76 (doi: 10.26583/sv.15.4.06)
4. Mikhaylyuk M.V., Kononov D.A., Loginov D.M. Modeling Situations in Virtual Environment Systems // Proceedings of the 23rd Conference on Scientific Services & Internet. – 2021. – Vol. 3066. – pp. 173–181 (doi: 10.20948/abrau-2021-6s-ceur)
5. Maltsev A.V., Strashnov E.V., Mikhaylyuk M.V. Methods and Technologies of Cosmonaut Rescue Simulation in Virtual Environment Systems // Scientific Visualization. – 2021. – Vol. 13, No. 4. – pp. 52–65 (doi: 10.26583/sv.13.4.05)
6. Strashnov E.V., Mironenko I.N. Simulation of hydraulic actuator dynamics in virtual environment systems // Software & Systems. – 2023. – Vol. 36, No. 4. – pp. 582–589 [in Russian] (<https://swsys.ru/files/2023-4/582-589.pdf>)
7. Bogolepov D.K., Sopin D.P., Ulyanov D.Ya., Turlapov V.E. Interactive GPU simulation of global illumination in animated heterogeneous scenes // Vestnik of Lobachevsky University of Nizhni Novgorod. – 2012. – No. 5 (2). – pp. 253–261 [in Russian] (<https://elibrary.ru/pzkqdz>)
8. 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, No 4. – pp. 243–249 (doi: 10.1134/S0361768817040065)
9. Maltsev A.V. Computer simulation and visualization of wheel tracks on solid surfaces in virtual environment // Scientific Visualization. – 2023. – Vol. 15, No. 2. – pp. 80–89 (doi: 10.26583/sv.15.2.07)
10. Chentanez N., Müller M. Real-time Simulation of Large Bodies of Water with Small Scale Details // Eurographics / ACM SIGGRAPH Symposium on Computer Animation (Eds. Otaduy M., Popovic Z.). – 2010. – pp. 197–206 (doi: 10.2312/SCA/SCA10/197-206)
11. NVIDIA RTX platform, NVIDIA Developer. (<https://developer.nvidia.com/rtx/>)
12. Timokhin P.Y., Mikhaylyuk M.V. An Efficient Technology of Real-time Modeling of Height Field Surface on the Ray Tracing Pipeline // Programming and Computer Software. – 2023. – Vol. 49, No. 3. – pp. 178–186 (doi: 10.1134/S0361768823030064)
13. Timokhin P.Y., Mikhaylyuk M.V. A Method to Order Point Clouds for Visualization on the Ray Tracing Pipeline // Programming and Computer Software. – 2024. – Vol. 50, No 3. – pp. 264–272 (doi: 10.1134/S0361768824700075)
14. Segal M., Akeley K. The OpenGL Graphics System: A Specification (Version 4.6 (Core Profile) - May 5, 2022) // The Khronos Group. – 2006–2022 (<https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf>)
15. Leese G., Kessenich J., Baldwin D., Rost R. The OpenGL Shading Language, Version 4.60.8 (14 Aug 2023) // The Khronos Group. – 2008–2022 (<https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf>)
16. Daniell P., Hector T., Olson T., Fredriksen J.-H, Trevett N., Potter R., Nguyen H., Ghamvam K., Caloca R. Vulkan, OpenGL and OpenGL ES // Khronos Group, SIGGRAPH 2017 (https://www.khronos.org/assets/uploads/developers/library/2017-siggraph/o6_3D-BOF-SIGGRAPH_Aug17.pdf)
17. Vulkan 1.3.283 - A Specification (with all ratified extensions) // The Khronos Group. – 2014–2024 (<https://registry.khronos.org/vulkan/specs/1.3-khr-extensions/pdf/vkspec.pdf>)
18. Frolov V., Sanzharov V., Galaktionov V., Scherbakov A. An Auto-Programming Approach to Vulkan // Proceedings of the 31th International Conference on Computer Graphics and Vision (GraphiCon 2021). – 2021. – Vol. 3027. – pp. 150–165 (doi: 10.20948/graphicon-2021-3027-150-165)

19. Unterguggenberger J., Kerbl B., Wimmer M. Vulkan all the way: Transitioning to a modern low-level graphics API in academia // *Computers & Graphics*. – 2023. – Vol. 111. – pp. 155-165 (doi: 10.1016/j.cag.2023.02.001)
20. NVIDIA Turing GPU Architecture // NVIDIA Corporation. – 2018. (<https://images.nvidia.com/aem-dam/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>)
21. Shirley P., Morley R.K. Realistic Ray Tracing, Second Edition // A K Peters, Natick, Massachusetts. – 2003. – 235 pages
22. Frolov V.A., Voloboy A.G., Ershov S.V., Galaktionov V.A. Light Transport in Realistic Rendering: State-of-the-Art Simulation Methods // *Programming and Computer Software*. – 2021. – Vol. 47, No 4. – pp. 298-326 (doi: 10.1134/S0361768821040034)
23. NVIDIA Ada GPU Architecture // NVIDIA Corporation. – 2022. (<https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf>)
24. Rusch M., Bickford N., Subtil N. Introduction to Vulkan Ray Tracing // *Ray Tracing Gems II*, NVIDIA. – 2021. – pp. 213-255 (doi: 10.1007/978-1-4842-7185-8_16)
25. *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX* // Ed. by Marrs A., Shirley P., Wald I. – Apress. – 2021. (doi: 10.1007/978-1-4842-7185-8)
26. Sanzharov V.V., Frolov V.A., Galaktionov V.A. Survey of Nvidia RTX Technology // *Programming and Computer Software*. – 2020. – Vol. 46, No. 4. – pp. 297-304 (doi: 10.1134/S0361768820030068)
27. Timokhin P.Yu., Mikhaylyuk M.V. Hybrid Visualization with Vulkan-OpenGL: Technology and Methods of Implementation in Virtual Environment Systems // *Scientific Visualization*. – 2023. – Vol. 15, No. 3. – pp. 7-17 (doi: 10.26583/sv.15.3.02)
28. Lefrancois M.-K., OpenGL Interop, NVIDIA DesignWorks Samples (https://github.com/nvpro-samples/gl_vk_simple_interop)
29. GL_EXT_memory_object, OpenGL Vendor and EXT Extension Specifications. 2018, (https://registry.khronos.org/OpenGL/extensions/EXT/EXT_external_objects.txt)
30. glDepthRange, OpenGL 4.5 Reference Pages. 2010-2014 Khronos Group, (<https://registry.khronos.org/OpenGL-Refpages/gl4/html/glDepthRange.xhtml>)
31. Song H.A., OpenGL Transformation. 2008-2021, (https://songho.ca/opengl/gl_transform.html)
32. Song H.A., OpenGL Projection Matrix. 2019, (https://songho.ca/opengl/gl_projectionmatrix.html)
33. Normalized Integer, 2018 (https://www.khronos.org/opengl/wiki/Normalized_Integer)
34. «Beautiful Autumn», a 3D Model by Epic_Tree_Store // Sketchfab (<https://skfb.ly/VrYw>)