Scientific Visualization, 2023, volume 15, number 3, pages 7 - 17, DOI: 10.26583/sv.15.3.02

Hybrid Visualization with Vulkan-OpenGL: Technology and Methods of Implementation in Virtual Environment Systems

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

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

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

Abstract

In this paper, the topic of integrating visualization tasks to be solved using the Vulkan API into virtual environment systems based on OpenGL visualization, is researched. The problem of Vulkan-OpenGL hybrid visualization and an approach to its solution, based on a modified render-to-texture technique, are described. The technology of constructing an original embeddable program shell (VK-capsule) is proposed, which allows hybrid visualization as a "black box" to be worked with, obtaining an image in the OpenGL frame buffer at the output. The paper presents the developed structure of the VK-capsule, comprising three program blocks (VK-, GL- and I-block), and describes methods and algorithms for their construction. Based on proposed technology, methods and algorithms, a VK-capsule for height field visualization task was developed, which utilizes hardware-accelerated ray tracing, Vulkan API supported by. The approbation of the developed VK-capsule was carried out, which showed that proposed solutions are effective and meet the task. The results obtained can be used in virtual environment systems, scientific visualization, video simulators, virtual laboratories, educational applications, etc.

Keywords: virtual environment systems, Vulkan, OpenGL, interoperability, hybrid visualization, library, interface, shared video memory, shared semaphore.

1. Introduction

Currently, open, platform-independent programming interfaces (APIs) become increasingly relevant in the field of real-time 3D computer graphics. One of these is the well-known OpenGL standard [1], which is in demand in modern virtual environment systems (VES) [2, 3], scientific visualization [4, 5], simulators [6], etc. However, despite the intuitive interface and thoughtful architecture, this API has a number of ideological limitations restraining its further development [7]. Therefore, the Khronos Group industrial consortium, the OpenGL standard is supervised by, launched the development of the next-generation graphics and computing API - Vulkan [8]. As OpenGL, this API is open and cross-platform, but it has lower overhead costs when processing calculations, and also provides much deeper control over the GPU and less CPU load.

From the advantages of Vulkan its main drawback is followed - a low-level interface. Many routine tasks, that in OpenGL the driver is responsible for (GPU memory management, command queueing, frame buffering, etc.), in Vulkan must be implemented by a developer, which significantly complicates graphics programming. Various solutions have been proposed to get around this problem: the V-EZ auxiliary shell [9], pattern-based development automation [10], the VulkanSceneGraph wrapper library [11], etc. These solutions are united by application development implementing entirely on Vulkan, which is not always allowable. In particular, this applies to VES having a developed OpenGL-based visualization subsystem (GL-visualizer), interconnected with control and dynamics calculation subsystems [12]. In such cases, it is reasonable to single out separate subtasks that are solved more efficiently by means of new graphics technologies supported by Vulkan (for example, hardware-accelerated ray tracing [13]) and integrate them into the GL-visualizer.

In this paper, the technology and methods to solve this task are proposed, based on the construction of an original program shell implementing the combination of Vulkan and OpenGL visualization with small-invasive intrusion into the GL-visualizer. Section 2 discusses the problem of combining Vulkan and OpenGL visualization. Section 3 describes proposed technology and methods for implementing hybrid visualization. Section 4 presents results of approbation of the proposed technology exemplified on height field visualization task [14].

2. The problem of combining OpenGL- and Vulkan-visualization

Initially, Vulkan was positioned as the successor of OpenGL (glNext), however, ultimately, this API received a visualization ideology differing from its predecessor. In OpenGL, visualization is based on the *rendering context* – a special shell that transmits commands to the GPU, synchronizes them, and also communicates with the application window. In Vulkan ideology there is no such an auxiliary shell, and its functions are performed by a number of abstract objects (instance, devices, command queues, etc.), the configuration and synchronization of which is the responsibility of the developer. Such fundamental differences don't allow to do OpenGL- and Vulkan-visualization directly in the same application window, but also don't exclude the ability of collaboration and interaction of both APIs (*interoperability*).

The Vulkan-OpenGL interoperability implementation mechanism is shown in the examples of NVidia [15, 16] and Khronos Group [17]. This mechanism is based on extensions[1] released by the Khronos Group consortium:

- **for OpenGL:** EXT_external_objects (GL_EXT_memory_object, GL_EXT_semaphore), EXT_external_objects_fd (GL_EXT_memory_object_fd, GL_EXT_semaphore_fd), EXT_external_objects_win32 (GL_EXT_memory_object_win32, GL_EXT_semaphore_win32) [18];
- **for Vulkan:** VK_KHR_external_memory, VK_KHR_external_memory_capabilities, VK_KHR_external_memory_fd, VK_KHR_external_memory_win32, VK_KHR_external_semaphore, VK_KHR_external_semaphore_capabilities, VK_KHR_external_semaphore_fd, VK_KHR_external_semaphore_win32 [19].

These extensions introduce new types of objects: 1) *shared video memory* through which Vulkan and OpenGL can exchange data, and 2) *shared semaphore* - a primitive for synchronizing access of both APIs to shared resources (GPU and shared video memory). The new objects make the problem of collaborative visualization to be worked around by means of a *hybrid approach*, in which Vulkan-visualization is performed into a texture, and on the OpenGL side this texture is rendered to the entire screen.

The examples [15, 16] show the implementation of a hybrid approach based on the "all in one" principle, in which OpenGL- and Vulkan-visualization are closely intertwined in one application. This principle allows the feasibility of a hybrid approach to be demonstrated, however, it is not suitable for embedding in modern VESs with a modular architecture. In the technology, proposed in this paper, hybrid visualization is implemented in a separate program module with a high-level interface, which is connected to the GL-visualizer with minimal intrusion.

[1] A specification containing a description of new functions and constants that extend the capabilities of the standard's core. Extensions that have passed comprehensive testing and confirmed the stability of their work are added to the core of the new version of the standard.

3. Hybrid visualization implementation technology

In Vulkan ideology, any visualization (both on- and off-screen) is performed by means of render-to-texture technique. Suppose, there is an application performing rendering of some virtual environment into the texture using Vulkan API (hereinafter *VK-rendering*). In this paper, the task of embedding VK-rendering into the GL-visualizer of VES is considered. The proposed technology is based on the construction of an embeddable program shell that "encapsulates" VK-rendering and the hybrid approach (hereinafter *VK-capsule*). From the point of the GL-visualizer, the VK-capsule is a "black box" that accepts data as input through an agreed set of high-level interface functions, and outputs an image in the OpenGL frame buffer (see Figure 1).

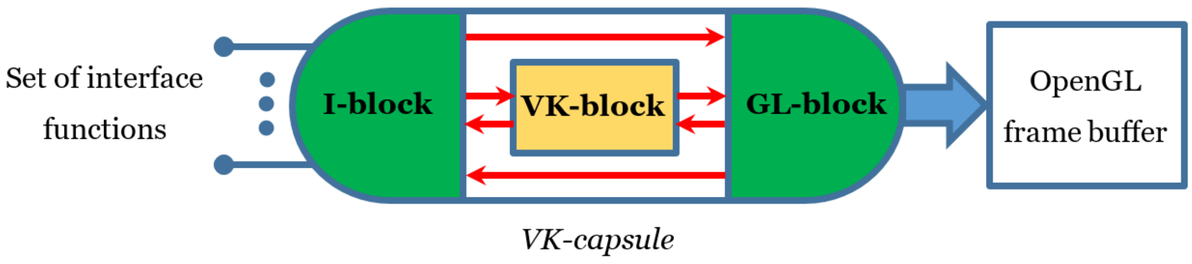


Fig. 1. VK-capsule structure.

The technology of VK-capsule construction is implemented in a standalone library module, dynamically linked to the GL-visualizer, and includes three stages: 1) construction of ***VK-block*** implementing Vulkan-side of hybrid visualization; 2) construction of ***GL-block*** implementing OpenGL-side of hybrid visualization; 3) construction of ***I-block*** implementing an interface for connecting the GL-visualizer to the VK-capsule module (VK- and GL-block). Let's consider the methods of implementing these stages.

3.1. Method of VK-block construction

Let's introduce the *RTT-core* term - a set of data structures and algorithms implementing VK-rendering. The proposed method is based on creation of a pair of classes: RTT-core "context" (*CVkContext*) and, derived from it, RTT-core "task" (*CVkTask*). The method is implemented in three phases, let's consider them in more detail.

At **the first phase (transferring)**, the RTT-core components are transferred from the source application to *CVkContext* and *CVkTask* classes. The basic set of Vulkan objects (*VkInstance*, *VkDevice*, *VkQueue*, etc.) needed to implement rendering into a texture is transferred to the *CVkContext* class. A more detailed description of such objects can be found in [20]. The basic set, in particular, includes objects used for implementation of: a) screen texture into which VK-rendering is performed (hereinafter *RTT-texture*) and b) synchronization of access to RTT-texture and GPU. In this paper, the following designations are introduced for these objects:

- *rttImg* - texture object (*VkImage*) that stores RTT-texture parameters;
- *rttMem* - video memory area (*VkDeviceMemory*) with RTT-texture image;
- *rttSems*[2] - a pair of semaphores (*VkSemaphore*), where *rttSems*[0] is a semaphore of RTT-texture readiness for updating, *rttSems*[1] is a semaphore of completing RTT-texture synthesis on the GPU.

Data structures and algorithms related directly to the VK-rendering task are transferred to the *CVkTask* class (see an example of such components in [21]). Note, that as a result of the first phase, the *CVkTask* class should include three methods: *init*, *deinit* and *render* - initialization, deinitialization (returning to the state before initialization) and synthesis of RTT-texture of the VK-rendering task.

At **the second phase (modification)**, *rttImg*, *rttMem* and *rttSems* objects are modified for hybrid visualization. In particular, the ability to work with external (exportable) memory is activated for the *rttImg* object, and *rttMem* and *rttSems* objects are made shared (see Section 2). This is implemented by the following algorithm[2]:

1. Before creating Vulkan instance (*VkInstance*), add the VK_KHR_external_memory-_capabilities and VK_KHR_external_semaphore_capabilities extensions to the *VkInstanceCreateInfo* structure.
2. Before creating Vulkan logical device (*VkDevice*), add the VK_KHR_external_memory, VK_KHR_external_memory_win32, VK_KHR_external_semaphore, VK_KHR_external_semaphore_win32 extensions to the *VkDeviceCreateInfo* structure.
3. Before creating *rttImg* texture object, fill the *VkExternalMemoryImageCreateInfo* structure and add it to the *VkImageCreateInfo* structure.
4. Before creating *rttMem* video memory area, fill the *VkExportMemoryAllocateInfo* structure and add it to the *VkMemoryAllocateInfo* structure.
5. Before creating *rttSems* semaphores, fill the *VkExportSemaphoreCreateInfo* structure and add it to the *VkSemaphoreCreateInfo* structure of each of *rttSems*[*i*]-th semaphore.

Note, that before executing step 1 of the algorithm, it is necessary to make sure that the extensions being added are in the list of supported extensions of the Vulkan instance (see the function *vkEnumerateInstanceExtensionProperties* [8]). For step 2 of the algorithm, it is necessary to perform a similar check using the *vkEnumerateDeviceExtensionProperties* function.

At **the third phase (extension)**, the interoperability interface (hereinafter *IOP-interface*) is implemented, by means of which the OpenGL API gets access to modified RTT-core objects, in particular, to shared *rttMem* and *rttSems* objects. The proposed IOP-interface is based on the following data structure:

$$\begin{array}{l} SIopInterface \\ \{ \\ \quad HANDLE \quad hRttMem; \\ \quad uint64\_t \quad rttMemSize; \\ \quad HANDLE \quad hRttSems[2]; \\ \}; \end{array} \qquad (1)$$

where *hRttMem* is the descriptor of the shared *rttMem* video memory area; *rttMemSize* is the size of the shared *rttMem* video memory area; *hRttSems*[2] are descriptors of shared *rttSems* semaphores.

Filling the *SIopInterface* structure is implemented in the *exportIop* method of the *CVkContext* class using data structures and functions introduced by platform-dependent Vulkan API extensions from Section 2. The filling algorithm comprises the following steps:

1. To get the value of the *hRttMem* field:
   - fill the fields of the *VkMemoryGetWin32HandleInfoKHR* structure: *memory* = *rttMem*, the rest fields - according to the Vulkan specification [8].
   - get the *hRttMem* descriptor using the *vkGetMemoryWin32HandleKHR* function and the filled *VkMemoryGetWin32HandleInfoKHR* structure.
2. Get the value of the *rttMemSize* field using the *VkMemoryRequirements* structure, the *rttImg* object and the *vkGetImageMemoryRequirements* function.
3. To get the value of *hRttSems*-th field:

   Loop by *hRttSems*[*i*]-th descriptors
   - fill the fields of the *VkSemaphoreGetWin32HandleInfoKHR* structure: *semaphore* = *rttSems*[*i*], the rest fields - according to the Vulkan specification [8];
   - get the *hRttSems*[*i*] descriptor using the *VkSemaphoreGetWin32HandleInfo-KHR* structure and the *vkGetSemaphoreWin32HandleKHR* function.

[2] Descriptions of the data structures given in the algorithm can be found in the Vulkan specification [8].

The above algorithm completes the third phase of the VK-block construction. As a result of the execution of all three phases, a pair of classes *CVkContext* and *CVkTask*, implementing the VK-block, are formed. Note, that the obtained *CVkContext* class is universal, which allows the construction of other VK-blocks (VK-capsules) to be significantly simplified.

3.2. Method of GL-block construction

The GL-block under consideration in this section implements: the representation of RTT-texture in OpenGL context (hereinafter *GL-texture*), synchronization of OpenGL with Vulkan and rendering GL-texture into OpenGL frame buffer. To solve these tasks, RTT-texture visualizer class (*CRttVisualizer*) is created, which includes the following number of key OpenGL-objects

- *glTex*, *glMem* - a pair of objects implementing GL-texture, where *glTex* is texture object storing GL-texture parameters, and *glMem* is video memory area with GL-texture image;
- *glSems*[2] - representations of a pair of *rttSems* semaphores (see Section 3.1) in OpenGL context (hereinafter *GL-semaphores*);
- *rectVao* - a vertex array object (VAO) that stores a polygonal model of a screen rectangle (positions and texture coordinates of vertices);
- *rectRenderer* - a shader program that renders textured *rectVao* model of screen rectangle into OpenGL frame buffer.

Based on the given member data, four key methods are implemented in the *CRttVisualizer* class: *init* and *deinit* - initialization and deinitialization of RTT-texture visualizer, as well as *prerender* and *postrender* - methods called before and after VK-block's *render* method (see Section 3.1). Let's consider them in more detail.

The *init* and *deinit* methods. The task of the *init* method is to create the OpenGL-objects listed above in video memory. GL-texture and GL-semaphore objects are created using OpenGL extension functions from Section 2 and the filled object of the structure (1) (hereinafter the *iop* object) passed to the *init* method as an argument. This implements the following algorithm

1. For GL- texture:
    - create *glMem* object of video memory area using the *glCreateMemoryObjectsEXT* function;
    - associate *glMem* object with shared *rttMem* video memory area of RTT-texture (see Section 3.1) using its descriptor *iop.hRttMem*, size *iop.rttMemSize* and the *glImportMemoryWin32HandleEXT* function;
    - create *glTex* texture object using the *glCreateTextures* function;
    - bind *glTex* texture object to *glMem* video memory area object using the *glTextureStorageMem2DEXT* function.
2. For GL- semaphores:

Loop by *glSems*[*i*]-th objects

- create *glSems*[*i*]th object using the *glGenSemaphoresEXT* function;
- associate *glSems*[*i*]th object with shared *rttSems*[*i*]th semaphore (see Section 3.1) using its descriptor *iop.hRttSems*[*i*] and the *glImportSemaphoreWin32HandleEXT* function.

The construction of *rectVao* model and *rectRenderer* shader program is typical and includes the creation of a pair of vertex buffer objects (VBO) storing positions and texture coordinates of screen rectangle vertices, as well as a pair of shader subroutines (vertex and fragment).

The task of the *deinit* method is to delete the OpenGL objects listed above from the video memory. As with the creation, OpenGL extension functions from Section 2 are used to delete GL-texture and GL-semaphores. In particular, the *glDeleteMemoryObjectsEXT* function is used for *glMem* object, and *glDeleteSemaphoresEXT* - for *glSems* objects.

The *prerender* and *postrender* methods work in conjunction with the VK-block's *render* method and are used to synthesize a hybrid visualization frame. To solve this task, it is necessary to synchronize the access of two APIs to shared resources (GPU and video memory). If this is not done, the APIs will interfere with each other (for example, Vulkan is still calculating RTT-texture, and OpenGL is already starting to render it on the screen), and the result will be unpredictable. Synchronization is implemented using the GL-semaphores introduced above, which can only be switched via the GPU to the signal or standby state (default value). GL-semaphores are controlled by both APIs: on the Vulkan side - through *rttSems* objects, on the OpenGL side - through *glSems* objects. Keeping this in mind, the following algorithm of hybrid visualization frame synthesis is formed

1. GL-block (the *prerender* method):
   - set *glSems*[0] object to the signal state (RTT-texture readiness for updating) using the *glSignalSemaphoreEXT*.
2. VK-block (the *render* method):
   - wait for a signal from *rttSems*[0] object;
   - perform RTT-texture synthesis;
   - set *rttSems*[1] object to the signal state (completing RTT-texture synthesis on the GPU).
3. GL -block (the *postrender* method):
   - wait for a signal from *glSems*[1] object using the *glWaitSemaphoreEXT* function;
   - make current the *glTex* texture object of GL-texture;
   - visualize the *rectVao* model of a screen rectangle, covered with GL-texture, using the *rectRenderer* shader program.

Note, that in steps 2 and 3 of the algorithm, after receiving signals, *rttSems*[0] and *glSems*[1] objects automatically reset GL-semaphores to the standby states, which allows them to be reused on the next frame of hybrid visualization.

3.3. Method of I-block construction

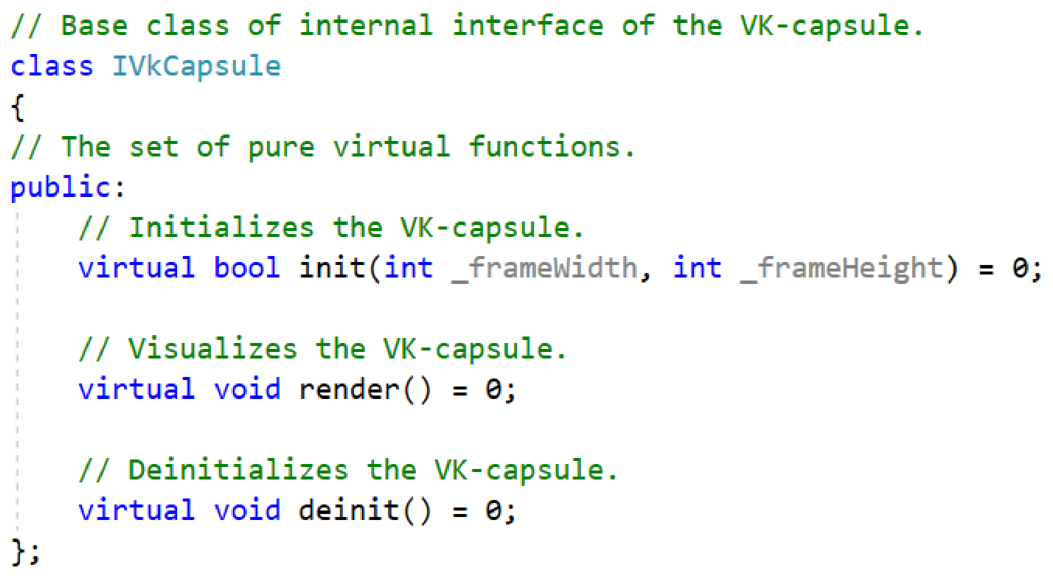
As noted in Section 3, the I-block implements the interface for connecting the GL-visualizer to VK-capsule library module. The proposed method of I-block construction includes the creation of two interfaces: the *internal*, the VK-capsule operation is controlled via, and the *external* (*exported*), the GL-visualizer and the VK-capsule module are communicated via. Let's consider them in more detail.

The **internal interface** is created based on a pair of classes: interface specification (*IVkCapsule*) and, derived from it, interface implementation class (*CVkCapsule*). The *IVkCapsule* class is abstract and contains only a set of pure virtual functions of the internal interface (see the example in Figure 2). The *CVkCapsule* class contains implementations of these virtual functions.

The set of virtual functions of the internal interface comprises the basic and user parts. The basic part includes the functions of initialization, visualization and deinitialization of the VK-capsule:

- *init* - allocates the memory for objects of the *CVkTask* and *CRttVisualizer* classes (hereinafter *pVkBlock* and *pGlBlock* objects), initializes these objects and interconnects them via the IOP-interface (see Sections 3.1, 3.2);
- *render* - implements the algorithm of hybrid visualization frame synthesis from Section 3.2 using *pVkBlock* and *pGlBlock* objects;
- *deinit* - deinitializes the *pVkBlock* and *pGlBlock* objects and frees the memory allocated for them.

The user part comprises interface functions of the VK-rendering task (the *CVkTask* class, see Section 3.1). In general, the list of such functions is quite extensive and includes functions for setting positions and orientations of scene objects; controlling virtual camera and light sources; processing window, keyboard, mouse events, etc.



```
// Base class of internal interface of the VK-capsule.
class IVkCapsule
{
// The set of pure virtual functions.
public:
    // Initializes the VK-capsule.
    virtual bool init(int _frameWidth, int _frameHeight) = 0;

    // Visualizes the VK-capsule.
    virtual void render() = 0;

    // Deinitializes the VK-capsule.
    virtual void deinit() = 0;
};
```

Fig. 2. An example of *IVkCapsule* base class of the internal interface.

The **external interface** is created based on a pair of exportable global functions

$$\begin{array}{l} \_\_declspec(dllexport)\ IVkCapsule{*}\ vkCapsuleCreate(); \\ \_\_declspec(dllexport)\ void\ vkCapsuleDestroy(IVkCapsule{**}\ \_ppVkCapsule); \end{array} \qquad (2)$$

where *vkCapsuleCreate*, *vkCapsuleDestroy* are functions for creating and destroying an object of the *CVkCapsule* class, and *__declspec(dllexport)* is a keyword pointing out the compiler that the function will be exported from the library.

Binding of the external interface to the internal interface is carried out inside the *vkCapsuleCreate* function when creating an object of the *CVkCapsule* class:

$$IVkCapsule{*}\ pVkCapsule = new\ CVkCapsule(); \qquad (3)$$

It can be seen from expression (3) that the *pVkCapsule* pointer to the created object of the *CVkCapsule* class has the type of the *IVkCapsule** base class. In fact, *pVkCapsule* is a pointer to virtual function table of the *IVkCapsule* base class, created at VK-capsule module compilation stage. After executing the expression (3), this table is filled by addresses of the corresponding virtual functions of the created object of the derived *CVkCapsule* class (the so-called dynamic polymorphism or late binding). As a result, a *pVkCapsule* pointer is formed, which provides access to all functions of the VK-capsule's internal interface.

As the result, the I-block comprising the pair of classes {*IVkCapsule*, *CVkCapsule*} and the pair of functions {*vkCapsuleCreate*, *vkCapsuleDestroy*} is created, which completes VK-capsule construction and adds a "gateway" to it to connect to the GL-visualizer. In this paper, such a connection is implemented using the WinAPI based on explicit dynamic linking of executable module of the GL-visualizer with VK-capsule library via developed external interface (2). Below, exemplified on a VK-capsule with basic internal interface, connection algorithm is shown:

1. At the stage of GL-visualizer initialization:
   - load VK-capsule library into GL-visualizer process address space using the *LoadLibrary* function;
   - get the addresses of the *vkCapsuleCreate* and *vkCapsuleDestroy* functions of the external interface of VK-capsule library using the *GetProcAddress* function;
   - create a *pVkCapsule* object of the internal interface of VK-capsule library using the *vkCapsuleCreate* function;
   - call the *init* method of the *pVkCapsule* object.
2. At the stage of GL-visualizer frame formation:
   - call the *render* method of the *pVkCapsule* object.

3. At the stage of GL-visualizer deinitialization:
   - call the *deinit* method of the *pVkCapsule* object;
   - destroy the *pVkCapsule* object using the *vkCapsuleDestroy* function;
   - unload VK-capsule library from GL-visualizer process address space using the *FreeLibrary* function.

Exemplified on the above algorithm, it can be seen that connecting the VK-capsule to the GL-visualizer is minimally invasive. In particular, to work with a VK-capsule, regardless of the task it solves, the addresses of only two functions (*vkCapsuleCreate* and *vkCapsuleDestroy*, see stage 1 of the algorithm) are required to obtain. This greatly simplifies the process of embedding VK-capsule into the GL-visualizer and their further maintenance.

4. Results and conclusions

Based on the proposed technology, methods and algorithms, a VK-capsule was developed that implements hybrid visualization of a detailed height field, based on hardware-accelerated ray tracing [14]. The VK-capsule is implemented in C++ using the GLSL shading programming language, Vulkan and OpenGL APIs. A prototype of GL-visualizer was also created, providing the creation of an OpenGL window and context, into which the developed VK-capsule was embedded. The created complex was tested, including hybrid visualization of the Puget Sound height field from the study [14]. Visualization was performed at Full HD screen resolution using NVidia GeForce RTX 2080 graphics card (Vulkan v. 1.3.204.1, NVidia DCH v. 512.59). During the visualization, interactive control of observer and light source motion was performed, as well as enabling/disabling the calculation of height field self-shadowing. Figure 3 shows an example of hybrid visualization of 8Kx8K Puget Sound height field with self-shadowing enabled. The conducted approbation confirmed the adequacy of the proposed technology to the task and the possibility of its effective application in software systems performing complex interactive visualization.

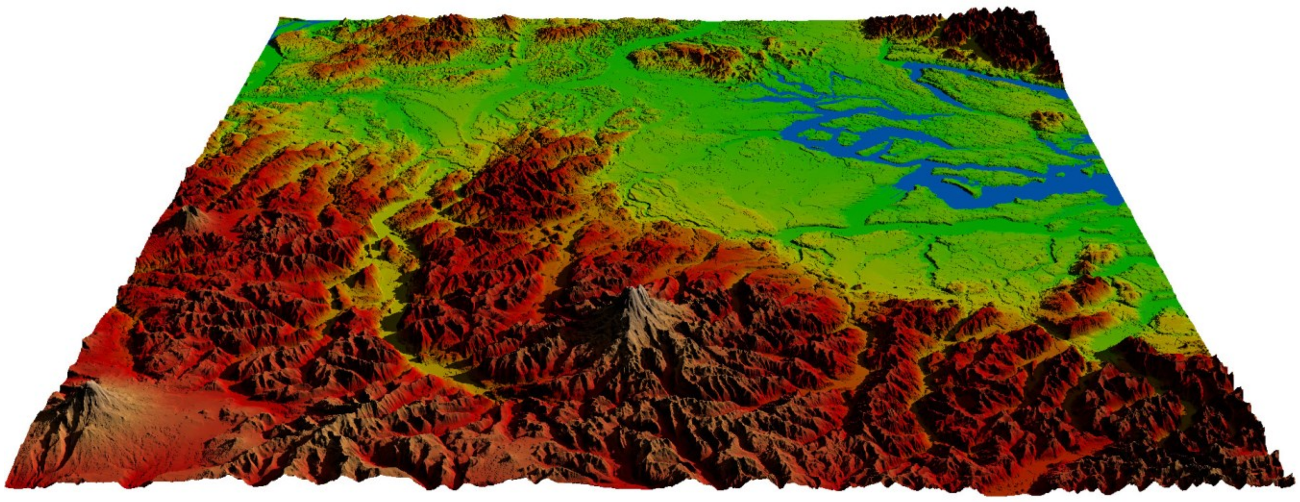


Fig. 3. An example of hybrid visualization of the Puget Sound height field using the developed technology.

As a result of the research, the following conclusions were obtained:

1. The proposed solution contains a number of universal components of hybrid visualization (*CVkContext* and *CRttVisualizer* classes, external interface (2)), which significantly simplify the construction of VK-capsules.

2. The proposed solution preserves the possibility of the GL-visualizer working without the VK-capsule library (due to the explicit dynamic linking of modules). This allows effective software package configurations for specific tasks (in particular, where only OpenGL visualization is needed) to be created, as well as develop modules independently of each other (in

the presence of the agreed internal interface of the VK-capsule). This is especially demanded when it is required to implement sophisticated modern graphics technologies in the VK-capsule, such as hardware-accelerated ray tracing.

3. Thanks to the developed external interface (2) of the VK-capsule, the need to implement on the GL-visualizer side the routine "import stuff" (creating pointers, obtaining addresses) for each function of the VK-capsule's internal interface is eliminated. This makes it comfortable to embed VK-capsules with an extensive internal interface and facilitates further maintenance and scaling of the software complex.

5. 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 “Carrying out basic scientific researches (47 GP)” on topic No. FNEF-2022-0012 “Virtual environment systems: technologies, methods and algorithms of mathematical modeling and visualization”.

References

1. Segal M., Akeley K. The OpenGL Graphics System: A Specification (Version 4.6 (Core Profile) - May 5, 2022) (https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf)

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)
(http://www.gctc.ru/media/files/Periodicheskie_izdaniya/ppk_2020_4_total_37/5_sta t.a_mihailuk_pq.pdf)

3. UNIGINE 2 Engine. Real-time 3D engine for enterprise and technology enthusiasts (https://unigine.com/)

4. Zakharova A.A., Korostelyov D.A., Podvesovskii A.G., Bondarev A.E., Galaktionov V.A. Generalized Computational Experiment State Analysis Using Three-Dimensional Visual Maps // Scientific Visualization. – 2022. – Vol. 14, No. 4. – pp. 12–23 (doi: 10.26583/sv.14.4.02) (http://sv-journal.org/2022-4/02/en.pdf)

5. ParaView. Solutions (https://www.paraview.org/solutions/)

6. Barladian B.Kh., Voloboy A.G., Shapiro L.Z., Deryabin N.B., Valiev I.V., Andreev S.V., Solodelov Yu.A., Galaktionov V.A. Safety critical visualization of the flight instruments and the environment for pilot cockpit // Scientific Visualization. – 2021. – Vol. 13, No. 1. – pp. 124–137 (doi: 10.26583/sv.13.1.09) (https://sv-journal.org/2021-1/09/en.pdf)

7. Shiraef J. An Exploratory Study of High Performance Graphics Application Programming Interfaces // Thesis for the Master's degree in Computer Science. – The University of Tennessee at Chattanooga. – 2016 (https://core.ac.uk/download/pdf/51197608.pdf)

8. Vulkan 1.3.238 - A Specification (with all registered Vulkan extensions), The Khronos Vulkan Working Group. – 2022 (https://www.khronos.org/registry/vulkan/specs/1.3-extensions/pdf/vkspec.pdf)

9. V-EZ API Documentation (https://gpuopen-librariesandsdks.github.io/V-EZ/)

10. 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) (https://ceur-ws.org/Vol-3027/paper14.pdf)

11. VulkanSceneGraph (https://github.com/vsg-dev/VulkanSceneGraph)

12. Mikhaylyuk M.V., Kononov D.A., Loginov D.M. Situational modeling in virtual environment systems // Scientific service on the Internet: proceedings of the XXIII All-Russian Scientific Conference (September 20-23, 2021, online). — Moscow: Keldysh Institute of Applied Mathematics named after M.V. Keldysh. – 2021. — pp. 236-243 [in Russian] (doi: 10.20948/abrau-2021-1) (https://keldysh.ru/abrau/2021/theses/1.pdf)

13. 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) (https://link.springer.com/article/10.1134/S0361768820030068)

14. 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)

15. Lefrancois M.-K., OpenGL Interop, NVIDIA DesignWorks Samples (https://github.com/nvpro-samples/gl_vk_simple_interop)

16. Lefrancois M.-K., OpenGL Interop - Raytracing, NVIDIA DesignWorks Samples (https://github.com/nvpro-samples/gl_vk_raytrace_interop)

17. Vulkan Samples, OpenGL interoperability, The Khronos Group. – 2020-2023. (https://github.com/KhronosGroup/Vulkan-Samples/tree/main/samples/extensions/open_gl_interop)

18. OpenGL Extensions (https://registry.khronos.org/OpenGL/extensions/EXT/)

19. Vulkan Extensions (https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/)

20. Overvoorde A. Vulkan Tutorial. – 2023. (https://vulkan-tutorial.com/resources/vulkan_ tutorial_en.pdf)

21. Lefrancois M.-K., Gautron P., Bickford N., Akeley D. NVIDIA Vulkan Ray Tracing Tutorial (https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/)