DOI: 10.15514/ISPRAS-2023-35(4)-6

Технология синтеза программных комплексов с гибридной визуализацией Vulkan-OpenGL

П.Ю. Тимохин, ORCID: 0000-0002-0718-1436 <*webpismo@yahoo.de*>
М.В. Михайлюк, ORCID: 0000-0002-7793-080X <*mix@niisi.ras.ru*>

ФГУ «ФНЦ Научно-исследовательский институт системных исследований РАН», 117218, Россия, г. Москва, Нахимовский просп., д. 36, к.1.

Аннотация. В данной работе рассматривается задача встраивания компьютерной визуализации, выполняемой с помощью API Vulkan, в программные комплексы, основанные на API OpenGL. Описывается низкоуровневый гибридный подход к реализации совместной работы двух API в рамках одного приложения, а также организация и синхронизация доступа к совместно используемым ресурсам. Предлагается технология «инкапсуляции» гибридного подхода в отдельном библиотечном модуле (VK-капсуле) с высокоуровневым интерфейсом, который динамически подключается к исполняемому модулю OpenGL-комплекса (GL-визуализатору). В работе описаны методы построения и подключения интерфейса VK-капсулы, обеспечивающие минимальное вмешательство в GL-визуализатор. На основе предложенных методов и технологии был разработан прототип модульного программного комплекса, реализующего гибридную визуализацию Vulkan-OpenGL. Была проведена апробация созданного комплекса, которая подтвердила адекватность предложенных решений поставленной задаче и возможность их использования для расширения возможностей систем визуализации, построенных на базе OpenGL.

Ключевые слова: визуализация; программирование; GPU; Vulkan; OpenGL; интерфейс; библиотека.

Для цитирования: Тимохин П.Ю., Михайлюк М.В. Технология синтеза программных комплексов с гибридной визуализацией Vulkan-OpenGL. Труды ИСП РАН, том 35, вып. 4, 2023 г., стр. 121–128. DOI: 10.15514/ISPRAS–2023–35(4)–6.

Благодарности: Публикация выполнена в рамках государственного задания ФГУ ФНЦ НИИСИ РАН “Проведение фундаментальных научных исследований (47 ГП)” по теме № FNEF-2022-0012 “Системы виртуального окружения: технологии, методы и алгоритмы математического моделирования и визуализации”.

A technology to synthesize software complexes with hybrid visualization Vulkan-OpenGL

P.Yu. Timokhin, ORCID: 0000-0002-0718-1436 <*webpismo@yahoo.de*>
M.V. Mikhaylyuk, ORCID: 0000-0002-7793-080X <*mix@niisi.ras.ru*>

Scientific Research Institute for System Analysis of the Russian Academy of Sciences (SRISA), build. 1, 36, Nakhimovskiy Avenue, Moscow, 117218, Russia.

Abstract. In this paper, the task of embedding computer visualization, performed using the Vulkan API, into OpenGL-based software complexes, is considered. A low-level hybrid approach to implement the collaboration of two APIs within the same application is described, as well as, the organization and synchronization of access to shared resources. The technology is proposed, which "encapsulates" the hybrid approach in a separate library module (VK-capsule) with a high-level interface that is dynamically linked to the executable module of

OpenGL-complex (GL-visualizer). The paper describes methods for construction of the interface and connection of the VK-capsule, providing minimal intrusion into GL-visualizer. Based on the proposed methods and technology, a prototype of modular software complex implementing hybrid Vulkan-OpenGL visualization was developed. The approbation of the created complex was carried out, which confirmed the adequacy of the proposed solutions to the task assigned and the possibility of using them to expand the capabilities of visualization systems built on the OpenGL.

Keywords: visualization; programming; GPU; Vulkan; OpenGL; interface; library.

For citation: Timokhin P.Yu., Mikhaylyuk M.V. A technology to synthesize software complexes with hybrid visualization Vulkan-OpenGL. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 4, 2023. pp. 121-128 (in Russian). DOI: 10.15514/ISPRAS-2023-35(4)-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 "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".

1. Введение

OpenGL [1] является одним из основных интерфейсов программирования (API) 3D компьютерной графики реального времени, востребованной в современных системах виртуального окружения [2], научной визуализации [3, 4], видеосимуляторах [5] и др. Являясь открытым и платформонезависимым стандартом, OpenGL имеет интуитивно понятный интерфейс и продуманную архитектуру, поддерживаемую всеми современными графическими картами, благодаря чему завоевал популярность среди разработчиков. Вместе с этим парадигма OpenGL имеет ряд особенностей, ограничивающих эффективность данного API и возможности реализации актуальных графических технологий, например, аппаратно-ускоренной трассировки лучей [6].

Ввиду этого еще в 2014 году промышленный консорциум Khronos Group (разработчик OpenGL) начал создание открытого, кроссплатформенного стандарта Vulkan [7] – графического и вычислительного API следующего поколения. Целью нового API стало снижение накладных расходов при выполнении графических операций, предоставление более полного контроля над работой GPU, а также уменьшение использования CPU. В этой связи API Vulkan стал более низкоуровневым (по сравнению с OpenGL), что в свою очередь привело к существенному росту трудоемкости программирования графики. Так, многие рутинные задачи, которые в OpenGL выполнял драйвер (формирование пула и очередей команд, буферизация кадров, управление памятью GPU и др.), в Vulkan разработчик должен явно прописывать в коде. Несколько облегчить разработку может применение библиотек-оберток Vulkan [8] или автоматизация на основе паттернов (шаблонов) [9]. Однако данные подходы ориентированы на то, что разработка приложения ведется полностью на Vulkan, что не всегда целесообразно. Зачастую, имея достаточно развитый функционал, основанный на OpenGL, в системе визуализации необходимо реализовать отдельные подзадачи, которые более эффективно решаются с помощью Vulkan (например, визуализация полей высот с помощью аппаратно-ускоренной трассировки лучей [10]). В этой ситуации возникает задача встраивания Vulkan-визуализации в программный комплекс, реализованный на OpenGL, и обеспечения их совместной работы на GPU.

Принципиальная возможность такой совместной работы (интероперабельность) показана в примерах, выпущенных компаниями NVidia [11, 12] и Khronos Group [13]. Недостатком данных материалов является формат «все в одном», характеризующийся тесным взаимным переплетением API OpenGL и Vulkan, обернутым в корпоративные фреймворки (системы классов), что существенно затрудняет понимание механизма практической реализации интероперабельности. Целью же данного исследования является разработка технологии, при которой Vulkan-визуализация выполняется в отдельном библиотечном модуле, подключаемом к OpenGL-приложению с минимальным вмешательством.

2. Подход к гибридной визуализации Vulkan-OpenGL

Несмотря на то, что Vulkan позиционируется как OpenGL следующего поколения, оба API имеют разные идеологии визуализации, в общем случае несовместимые друг с другом. В OpenGL передача команд на GPU и связь с окном приложения осуществляется через специальный объект (оболочку) - контекст отрисовки. В Vulkan понятия контекста отрисовки как такового нет, а его функции, по сути, распределены между рядом абстрактных объектов (экземпляр, физическое и логическое устройства, пулы, буферы и очереди команд и т.д.). И хотя принципиальные различия между контекстом OpenGL и «контекстом» Vulkan не позволяют проводить совместную визуализацию напрямую в окне приложения, это не исключает возможности работы обоих API в рамках одного программного комплекса [11]. Учитывая этот факт, обойти проблему совместной визуализации можно с помощью *гибридного подхода*, при котором Vulkan осуществляет отображение виртуального объекта (сцены) в текстуру (Render-to-texture, RTT-текстуру), а OpenGL визуализирует эту RTT-текстуру на весь экран. Реализация этого подхода включает решение следующих двух ключевых задач.

Во-первых, необходимо чтобы Vulkan и OpenGL могли «видеть» одну и ту же область видеопамяти и работать с ней, как с текстурой. Решение этой задачи основано на идее *разделяемого текстурного объекта*. Ее суть состоит в том, что на стороне Vulkan выделяется область видеопамяти, и к ней привязывается текстурный объект в «контексте» Vulkan, а на стороне OpenGL импортируется образ этой области видеопамяти (по сути, ее адрес и размер) и к нему привязывается текстурный объект в контексте OpenGL. Таким образом, мы фактически получаем два рабочих текстурных объекта, привязанные к одной области видеопамяти, в которой хранится RTT-текстура.

Во-вторых, необходимо синхронизировать доступ двух API к совместно используемым ресурсам (GPU и разделяемому текстурному объекту). Если этого не сделать, то API будут мешать друг другу (например, Vulkan еще досчитывает RTT-текстуру, а OpenGL уже выводит ее на экран), и результат будет непредсказуемым. Решение второй задачи основано на использовании *GL-семафоров* - специальных примитивов синхронизации, которые могут переключаться только посредством GPU в состояние сигнала или ожидания (значение по умолчанию). В рассматриваемой задаче используется пара GL-семафоров:

- *updateSemaphore* – готовность разделяемого текстурного объекта к обновлению;
- *synthesisSemaphore* – завершение синтеза RTT-текстуры на GPU.

Управление GL-семафорами осуществляется обоими API на стадии формирования кадра визуализации. Вначале сторона OpenGL переводит *updateSemaphore* в состояние сигнала, получив который сторона Vulkan начинает синтез RTT-текстуры в разделяемом текстурном объекте. По окончании синтеза RTT-текстуры сторона Vulkan переводит *synthesisSemaphore* в состояние сигнала, получив который сторона OpenGL начинает визуализацию RTT-текстуры на экране. Отметим, что после получения сигналов сторонами Vulkan и OpenGL *updateSemaphore* и *synthesisSemaphore* автоматически сбрасываются в состояние ожидания, что позволяет их снова использовать на следующем кадре визуализации. Чтобы реализовать описанное совместное управление синхронизацией, каждый GL-семафор создается *разделяемым*: на стороне Vulkan в видеопамяти создается сам объект семафора, а на стороне OpenGL импортируется образ этого объекта (по аналогии с разделяемым текстурным объектом).

Из описания гибридного подхода видно, что API Vulkan и OpenGL взаимодействуют на достаточно низком уровне, что существенно усложняет процесс встраивания Vulkan-визуализации в программные комплексы на базе OpenGL, имеющие свои сложившиеся экосистемы. В данной работе предлагается технология «инкапсуляции» гибридного подхода, позволяющая встраивать Vulkan-визуализацию в программные комплексы на базе OpenGL с минимальным вмешательством. Рассмотрим ее более подробно.

3. Технология «инкапсуляции» гибридного подхода

В данной работе мы будем рассматривать задачу встраивания программного блока, реализующего синтез RTT-текстуры с помощью API Vulkan (далее *VK-блок*), в исполняемый модуль, осуществляющий OpenGL-визуализацию (далее *GL-визуализатор*). Предлагаемое решение основано на построении *VK-капсулы* - программной оболочки, изолирующей работу VK-блока в отдельном библиотечном модуле, динамически связываемом с GL-визуализатором (см. рис. 1). VK-капсула является системой взаимосвязанных программных блоков, в которую кроме VK-блока входят блок гибридной визуализации (далее *HR-блок*) и интерфейсный блок (далее *I-блок*). HR-блок реализует отображение рассчитанной VK-блоком RTT-текстуры в буфере кадра GL-визуализатора с помощью гибридного подхода, описанного в разделе 2. I-блок реализует высокоуровневый программный интерфейс взаимодействия GL-визуализатора и VK-капсулы, и является ядром предлагаемой технологии «инкапсуляции» гибридного подхода. Рассмотрим ее этапы.

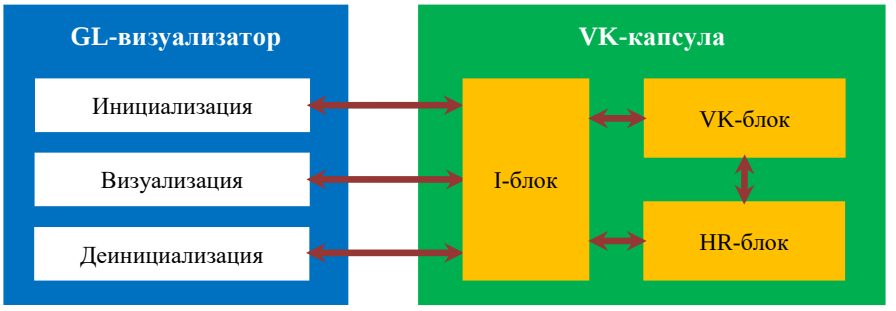

Рис. 1. Структура программного комплекса с гибридной визуализацией
Fig. 1. The structure of software complex with hybrid visualization

3.1 Построение интерфейса VK-капсулы

На данном этапе реализуется набор интерфейсных функций, посредством которых GL-визуализатор взаимодействует с VK- и HR-блоком. Набор состоит из базовой и пользовательской частей. В базовую часть входит минимальный список функций, необходимых для работы VK-капсулы:

- *init* - выделяет ресурсы, необходимые для VK- и HR-блока, инициализирует эти блоки, а также связывает их между собой через разделяемые GL-семафоры и текстурный объект (см. раздел 2);
- *render* - синтезирует RTT-текстуру с помощью VK-блока и отображает в буфере кадра GL-визуализатора с помощью HR-блока;
- *deinit* - освобождает ресурсы, выделенные для VK- и HR-блока и возвращает их в состояния до вызова функции *init*.

Пользовательская часть набора определяется, исходя из задачи, решаемой VK-блоком, и, в общем случае, может включать в себя достаточно большой список функций: задания позиций и ориентаций объектов сцены, управления виртуальной камерой и источниками освещения, обработки событий окна, клавиатуры, мыши и т.д.

Набор интерфейсных функций реализуется в I-блоке с помощью двух классов:

- *AVkCapsule* - абстрактный класс, состоящий из чистых виртуальных функций, описывающих набор интерфейсных функций VK-капсулы (см. рис. 2);
- *CVkCapsule* - наследуемый от *AVkCapsule* класс, который реализует его виртуальные функции (описанные выше связи с VK- и HR-блоком).

```
// Абстрактный класс интерфейса VK-капсулы.
class AVkCapsule
{
// == Набор интерфейсных функций VK-капсулы (чистых виртуальных функций) ==
// Базовая часть набора.
public:
    // Инициализирует VK-капсулу.
    virtual bool init(uint32_t _wndWidth, uint32_t _wndHeight, const char* _pScenePath) = 0;

    // Визуализирует VK-капсулу.
    virtual void render() = 0;

    // Деинициализирует VK-капсулу.
    virtual void deinit() = 0;

// Пользовательская часть набора.
public:
    // Обработчик события перемещения мыши.
    virtual void onMouseMove(float _xCoord, float _yCoord) = 0;

    // Обработчик события нажатия клавиши мыши.
    virtual void onMouseButton(int _buttonCode, int _actionType, int _mods) = 0;

    // Обработчик события нажатия клавиши клавиатуры.
    virtual void onKey(int _keyCode, int _scancode, int _actionType, int _mods) = 0;

    // Возвращает инфо-строку VK-капсулы.
    virtual const char* getInfoStr() = 0;
};
```

Рис. 2. Пример абстрактного класса AVkCapsule, включающего базовый и пользовательский набор интерфейсных функций

Fig. 2. An example of abstract AVkCapsule class including base and user sets of interface functions

3.2 Подключение VK-капсулы

На данном этапе реализуется *модифицированное* явное динамическое связывание библиотеки VK-капсулы и исполняемого модуля GL-визуализатора. В отличие от типового подхода, предполагающего экспорт всех интерфейсных функций библиотеки, в данной работе на стороне VK-капсулы (в I-блоке) экспортируется только пара функций:

- *create* - создает объект класса *CVkCapsule* в модуле VK-капсулы;
- *destroy* - удаляет объект класса *CVkCapsule*.

Ключевым аспектом является то, что при создании объекта *CVkCapsule* функция *create* возвращает на него указатель типа базового класса *AVkCapsule**:

$$AVkCapsule* \; pCapsule = new \; CVkCapsule(); \qquad (1)$$

Указатель *pCapsule*, по сути, представляет собой указатель на таблицу виртуальных функций класса *AVkCapsule*, созданную на этапе компиляции модуля VK-капсулы. После выполнения выражения (1), благодаря свойству полиморфизма программного кода, в эту таблицу будут записаны адреса соответствующих виртуальных функций объекта класса *CVkCapsule* (так называемое позднее связывание). Таким образом, имея лишь объявление базового класса *AVkCapsule* и указатель *pCapsule*, GL-визуализатор получает доступ ко всем интерфейсным функциям VK-капсулы.

Подключение VK-капсулы реализуется на трех типовых стадиях работы GL-визуализатора: инициализации, визуализации и деинициализации (см. рис. 1). Для Win-платформы алгоритм подключения VK-капсулы с базовым набором интерфейсных функций имеет следующий вид:

1. На стадии инициализации GL-визуализатора:

- загрузить модуль VK-капсулы с помощью функции *LoadLibrary*;

- получить адреса интерфейсных функций *create* и *destroy* модуля VK-капсулы с помощью функции *GetProcAddress*;
- получить указатель *pCapsule* на объект VK-капсулы с помощью функции *create*;
- вызвать через указатель *pCapsule* метод *init* объекта VK-капсулы.

2. На стадии визуализации (формирования кадра) GL-визуализатора вызвать через указатель *pCapsule* метод *render* объекта VK-капсулы.

3. На стадии деинициализации GL-визуализатора:

- вызвать через указатель *pCapsule* метод *deinit* объекта VK-капсулы;
- удалить объект VK-капсулы с помощью функции *destroy*;
- освободить модуль VK-капсулы от привязки к GL-визуализатору с помощью функции *FreeLibrary*.

4. Результаты

На основе предложенной технологии был разработан прототип модульного программного комплекса, реализующего гибридную визуализацию Vulkan-OpenGL. Программный комплекс написан на языке C++ с использованием языка GLSL программирования шейдеров. В комплекс входят GL-визуализатор, обеспечивающий создание окна и контекста OpenGL, и VK-капсула (построена на основе API Vulkan версии 1.3.204.1), реализующая гибридную визуализацию облаков точек с помощью аппаратно-ускоренной трассировки лучей. Была проведена апробация созданного комплекса на задаче моделирования и визуализации элементов рельефа с отрицательными уклонами (пещер, шахт, туннелей и др.). Для апробации использовался набор данных «NASA Planetary Pits and Caves Analog Dataset» [14], полученных на основе LIDAR-сканирования с высоким разрешением реальных наземных объектов. На рис. 3 показаны примеры кадров гибридной визуализации элемента рельефа «King's Bowl» («Королевская Чаша»), состоящего из 3740782 точек. Визуализация выполнялась со средней скоростью около 500 кадров в секунду на персональном компьютере, оборудованном видеокартой NVidia GeForce RTX 2080 (драйвер NVidia DCH версии 536.40), при разрешении экрана Full HD. Снижение производительности гибридной визуализации по сравнению с «чистой» Vulkan-визуализацией составило не более 8-10%.

5. Заключение

Проведенная апробация подтвердила адекватность предложенной технологии поставленной задаче и возможность ее эффективного применения в программных комплексах, выполняющих сложную интерактивную визуализацию, за счет следующих преимуществ.

Во-первых, VK-капсула и GL-визуализатор взаимодействуют напрямую, без использования промежуточной библиотеки импорта, создаваемой компоновщиком. Это позволяет разрабатывать модуль VK-капсулы в среде программирования, имеющей версию, отличную от используемой для GL-визуализатора. Это особенно актуально, когда в VK-капсуле (VK-блоке) необходимо реализовать современные графические технологии (например, аппаратно-ускоренную трассировку лучей), требующие использования сред программирования последних версий.

Во-вторых, благодаря явному типу связывания сохраняется возможность работы GL-визуализатора без библиотеки VK-капсулы (для задач, где необходима только OpenGL-визуализация). Это позволяет создавать эффективные конфигурации из модулей программного комплекса под конкретные задачи, а также вести разработку модулей независимо друг от друга (при наличии согласованного интерфейса VK-капсулы).

В-третьих, благодаря модификации явного связывания, на стороне GL-визуализатора устраняется необходимость реализации программной «обвязки» (создание указателей,

получение адресов) для каждой функции интерфейса VK-капсулы. Это особенно актуально, когда к GL-визуализатору необходимо подключить VK-капсулу с обширным набором интерфейсных функций. Также исключается конфликт имен интерфейсных функций при встраивании в GL-визуализатор нескольких VK-капсул с похожими интерфейсами. Все это делает процесс встраивания VK-капсулы комфортным и облегчает дальнейшее сопровождение и масштабирование программного комплекса.

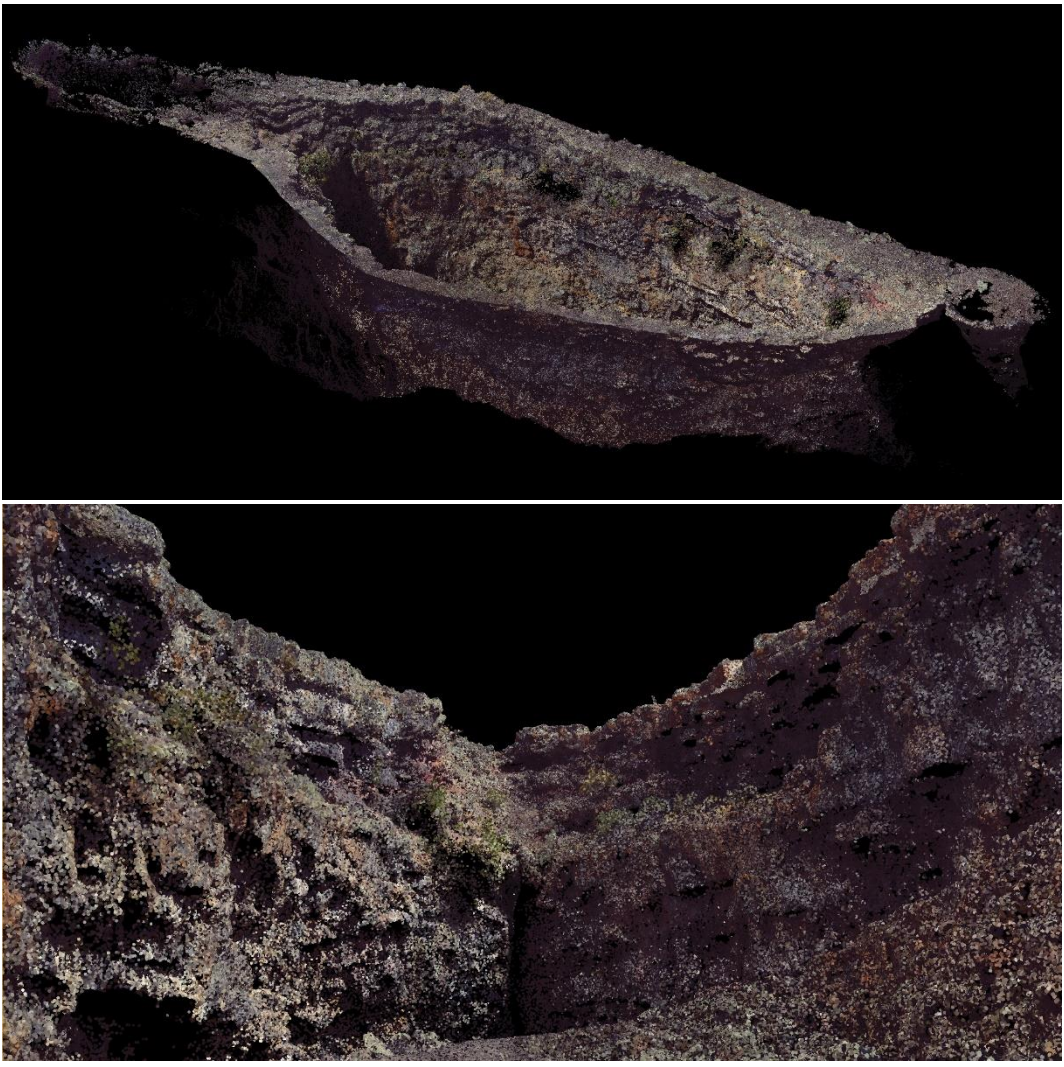

Рис. 3. Примеры Vulkan-визуализации элемента рельефа «King's Bowl» снаружи (вверху) и изнутри (внизу), выполняемой в OpenGL-окне с помощью разработанной технологии

Fig. 3. Examples of Vulkan-visualization of the "King's Bowl" terrain element from the outside (top) and from the inside (bottom), performed in OpenGL-window using the developed technology

Список литературы / References

[1]. OpenGL - The Industry Standard for High Performance Graphics. Available at: https://www.opengl.org/, accessed 13.07.2023.

[2]. Михайлюк М.В., Мальцев А.В., Тимохин П.Ю., Страшнов Е.В., Крючков Б.И., Усов В.М. Система виртуального окружения VirSim для имитационно-тренажерных комплексов подготовки

космонавтов. Пилотируемые полеты в космос, № 4(37), 2020 г., стр. 72-95. DOI: 10.34131/MSF.20.4.72-95. / Mikhaylyuk M.V., Maltsev A.V., Timokhin P.Yu., Strashnov E.V., Kryuchkov B.I., Usov V.M. The VirSim Virtual Environment System for the Simulation Complexes of Cosmonaut Training. Pilotiruemye polety v kosmos / Manned Spaceflight, № 4(37), 2020, pp. 72-95 (in Russian). DOI: 10.34131/MSF.20.4.72-95.

[3]. ParaView - Open-source, multi-platform data analysis and visualization application. Available at: https://www.paraview.org/, accessed 13.07.2023.

[4]. Avogadro - Free cross-platform molecule editor and visualizer. Available at: https://avogadro.cc/, accessed 13.07.2023.

[5]. UNIGINE: real-time 3D engine. Available at: https://unigine.com/, accessed 13.07.2023.

[6]. NVIDIA RTX platform. Available at: https://developer.nvidia.com/rtx, accessed 13.07.2023.

[7]. Vulkan - a cross-platform industry standard for 3D graphics and computing. Available at: https://www.vulkan.org/, accessed 13.07.2023.

[8]. VulkanSceneGraph (VSG), Vulkan & C++17 based Scene Graph Project. Available at: https://vsg-dev.github.io/vsg-dev.io/, accessed 13.07.2023.

[9]. Фролов В.А., Санжаров В.В., Галактионов В.А., Щербаков А.С. Автоматизация разработки на Vulkan: предметно-ориентированный подход. Труды ИСП РАН, том 33, вып. 5. 2021 г., стр. 181-204. DOI: 10.15514/ISPRAS-2021-33(5)-11 / Frolov V.A., Sanzharov V.V., Galaktionov V.A, Scherbakov A.S. Development in Vulkan: a domain-specific approach. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 5, 2021, pp. 181-204 (in Russian). DOI: 10.15514/ISPRAS-2021-33(5)-11.

[10]. Тимохин П.Ю., Михайлюк М.В. Рендеринг детализированных полей высот в реальном времени с использованием аппаратного ускорения трассировки лучей. Труды 32-й Международной конференции по компьютерной графике и машинному зрению (GraphiCon 2022), Рязань, 19-22 сентября 2022 г., стр. 124-135. DOI: 10.20948/graphicon-2022-124-135 / Timokhin P.Yu., Mikhaylyuk M.V. Real-time Rendering of Detailed Height Fields Using Hardware-based Ray Tracing Acceleration. In Proceedings of the 32th International Conference on Computer Graphics and Vision (GraphiCon 2022), Ryazan, Russia, September 19-22, 2022, pp. 124-135 (in Russian). DOI: 10.20948/graphicon-2022-124-135.

[11]. Lefrançois M.-K. OpenGL Interop. NVIDIA DesignWorks Samples. Available at: https://github.com/nvpro-samples/gl_vk_simple_interop, accessed 13.07.2023.

[12]. Lefrançois M.-K. OpenGL Interop – Raytracing. NVIDIA DesignWorks Samples. Available at: https://github.com/nvpro-samples/gl_vk_raytrace_interop, accessed 13.07.2023.

[13]. Vulkan Samples, OpenGL interoperability, The Khronos Group. 2020-2023. Available at: https://github.com/KhronosGroup/Vulkan-Samples/tree/main/samples/extensions/open_gl_interop, accessed 13.07.2023.

[14]. Wong U., Whittaker W., Jones H., Whittaker R. NASA Planetary Pits and Caves Analog Dataset. December 2014. Available at: https://ti.arc.nasa.gov/dataset/caves/, accessed 13.07.2023.

Информация об авторах / Information about authors

Петр Юрьевич ТИМОХИН – старший научный сотрудник ФГУ ФНЦ НИИСИ РАН. Сфера научных интересов: компьютерная графика, визуализация.

Petr Yurievich TIMOKHIN – Senior Researcher of SRISA RAS. Research interests: computer graphics, visualization.

Михаил Васильевич МИХАЙЛЮК – доктор физико-математических наук, профессор, главный научный сотрудник ФГУ ФНЦ НИИСИ РАН. Сфера научных интересов: компьютерная графика, визуализация, системы виртуального окружения.

Mikhail Vasilievich MIKHAYLYUK – Doctor of Physical and Mathematical Sciences, Professor, Chief Researcher of SRISA RAS. Research interests: computer graphics, visualization, virtual environment systems.