MULTITHREADED RENDERING FOR CROSS-PLATFORM 3D VISUALIZATION BASED ON VULKAN API

C. Ioannidis, A.-M. Boutsi*

Laboratory of Photogrammetry, School of Rural and Surveying Engineering, National Technical University of Athens, Greece;
cioannid@survey.ntua.gr, iboutsi@mail.ntua.gr

KEY WORDS: Computer graphics, 3D visualization, graphics API, Vulkan, geospatial data

ABSTRACT:

The visualization of large-sized 3D geospatial models is a graphics intensive task. With ever increasing size and complexity, more computing resources are needed to attain speed and visual quality. Exploiting the parallelism and the multi-core performance of the Graphics Processing Unit (GPU), a cross-platform 3D viewer is developed based on the Vulkan API and modern C++. The proposed prototype aims at the visualization of a textured 3D mesh of the Cultural Heritage by enabling a multi-threaded rendering pipeline. The rendering workload is distributed across many CPU threads by recording multiple command buffers in parallel and coordinating the host and the GPU rendering phases. To ensure efficient multi-threading behavior and a minimum overhead, synchronization primitives are exploiting for ordering the execution of queues and command buffers. Furthermore, push-constants are used to send uniform data to the GPU and render passes to adapt to the tile-based rendering of the mobile devices. The proposed methodology and technical solution are designed, implemented and tested for Windows, MacOS and Android on Vulkan-compatible GPU hardware by compiling the same codebase. The benchmarking on multiple hardware, architectures and platforms explores the performance improvement for the different approaches compared to one-thread and showcase the potential of the 3D viewer to handle large datasets at no expense of visual quality and geometric fidelity in the absence of high-end technological resources.

1. INTRODUCTION

In the fields of photogrammetry and topographic surveying, fast surface modelling techniques, range sensors and computer vision algorithms ensure the geometric fidelity and accuracy of their final products. The growth of multi-source and high-dimensional 3D spatial data availability intensifies demands for a dissemination strategy that clearly specifies their potential. Specific-domain knowledge such as diagnosis and restoration of Cultural Heritage, buildings and infrastructure plan and design of Building Information Modeling (BIM) or landscape and properties recording of 3D cadastral and GIS, can be easily diffused and interpreted through a dedicated 3D viewer. Therefore, the 3D visualization in a consistent and custom-oriented way, regardless of the operating system and hardware used, is becoming of the interest of cases where geospatial referencing is encountered as a crucial factor. Although the options for local rendering are numerous there is a little availability of software that handles portability, explicit control and high-performance at the same time. Furthermore, institutes, communities and research groups often lack of dedicated hardware and high-end processing units that provide responsiveness and a seamless visualization experience.

A solution for such visual applications derives from the low-level access to the GPU's architecture. Compared to the CPU, the GPU is equipped with more execution and memory units and specialized fixed-function chips that optimize its computing and memory capabilities (Wu et al, 2015). Over the last decade, its programmable functionality to compute and rendering operations has increasingly been supported by the majority of manufacturers. Graphics APIs expose this programmability on an abstraction level, transfer data and commands and ease the processes in all stages of computer graphics generation. Unlike traditional APIs like OpenGL, Vulkan API represents a closer mapping to the way GPUs are currently built. It uses an asynchronous rendering model in which, CPU and GPU synchronization, scheduling tasks order and device memory management are delegated to the application and defined by the developer. In order to attain Vulkan's performance boost, there is an obvious trade-off between flexibility in application structure and upfront development work on a more granular level. Despite its apparent complexity, Vulkan is widely supported due to extra benefits like precompiling shaders in SPIR-V format and multi-threading capabilities (Blackert, 2016). Prior research focuses mostly on General Purpose computing on the GPU (GPGPU) applications and scientific simulations (Gunadi and Yugopuspito, 2018; Thoman et al, 2020). The deployment of Vulkan and its multithreading capabilities to 3D visualization frameworks optimized for large-scale and complex geometry is less consistent.

Addressing this important deficiency, a cross-platform 3D model viewer with multithreading support is developed based on modern C++ and Vulkan API. It is suited to Windows, MacOS and Android and to every graphics hardware that offers Vulkan's driver support. The prototype application renders a 3D scene with a high-resolution textured mesh into an interactive User Interface (UI). The 3D mesh can be transformed with scaling, rotation and translation by the binding of mouse and touch screen events. The feature set includes Multi-Sampled Anti-Aliasing (MSAA) that alleviates geometrical aliasing, making geometrical edges look smoother and more temporarily stable. Key aspect of the development is the optimization of GPU and CPU performance, implementing the following methods:

- Multi-threaded command buffer generation with synchronization primitives

- Using of push-constants to send uniform data to the GPU
- Render passes for mobile GPU's tiled-rendering.

The proposed approach to the display of 3D geometry adapts to the implicit tile-based rendering of mobile GPUs, scaling from low-power mobile devices to high-end workstations. It is technologically innovative in terms of Vulkan API utilization, cross-vendor portability and performance. Many components can be re-used and the code can serve the needs of geospatial visualizations as a basis for a higher-level programming framework. The paper is structured as follows: Section 2 presents the literature review and how the proposed project progresses beyond the state-of-the-art, Section 3 analyzes the overall methodological approach of each objective presented in the Introduction and Section 4 describes the case study and their practical implementation until the formation of the technical solution. In Section 5, a performance evaluation is presented and finally, in Section 6, conclusions are drawn and an outlook of future research perspectives is provided.

2. RELATED WORK

GPUs have transformed to powerful graphics platforms coupled with a high parallel computing ability. The graphics APIs that determine the interaction of the application with their specialized code are evolving in accordance with the new standards. From the early days of computer graphics and the OpenGL API by Khronos to the lowest-level DirectX 12 by Microsoft, Vulkan by Khronos and Metal by Apple, their logic is converging to the structure and function of the modern GPUs. DirectX 12 runs solely on Windows and Xbox systems while Metal on Apple hardware and iOS. Therefore, for vendor- and platform-independency, graphics programming aligns towards OpenGL and Vulkan API. Ray-tracing integration, programmable rendering pipeline and shaders and multi-threading support constitute some of the recent hardware updates. The latter feature has been evolved over the past 20 years (Feinbube et al., 2011). Several techniques have been introduced to optimize multi-threading for highly intensive workloads like scheduling as a case management system (Rogers et al., 2014; Mittal, 2014) and memory transfer overhead reducing (Cho et al., 2019). The majority orients to computing operations performed by CUDA or OpenCL with GPGPU programming while the literature on multi-threading entirely for visual processes and 3D rendering is less consistent.

Over the past few years, a growing community has successfully mapped their specific domain applications onto the GPUs to exploit their parallel computation for graphics and non-graphics tasks (Owens et al., 2008). In the first case, the majority of the implementations concerns game engines (Grigg and Hexel, 2017; (Redlarski et al., 2018) and scientific simulations. Focusing on a subset of work conducted for geospatial data and geoinformation, both desktop and mobile applications will be presented. High resolution radar data were collected, converted and visualized at runtime on a GPU accelerated system with the support of OpenGL (Pezhgorski and Lazarova, 2017). A Level of Detail (LOD) method that explores the balance between visual quality and performance was proposed for Android devices exploiting the OpenGL API (Piao et al., 2014). On the same platform, the Vulkan API was used for 2D rendering of animations and effects and important performance gains were indicated concerning overhead and memory consumption (Gambhir et al., 2018). Fluid animation based on the SPH algorithm was simulated in GLSL compute shader in SPIR-V format in both Vulkan and OpenGL. The Vulkan implementation performed better compared to the OpenGL's one in the case of a high number of rendered particles (Gunandi and Yugopuspito, 2018). A rendering system for a large 3D model of the Berlin city has been developed with Vulkan API integrating a technique for streaming textures subsequently in order to reduce texture memory and optimize the overall performance (Zhang et al, 2018). A Vulkan abstraction layer that eases the implicit rendering configuration was developed tailored to large data is introduced by (Lavric et al, 2018) The higher-level interface that manages the object instances enables the remote visualization of large-sized data on lightweight client devices. The same codebase was compared with the equivalent CPU and OpenCL implementations and Vulkan was about 9 % faster. Regarding GIS and natural hazard risk assessment, a GPU-accelerated rendering pipeline was used to perform geospatial analysis methods to Big Data and visualize the results rather to web or mobile GIS applications (Heitzler et al., 2017). Finally, a Cultural Heritage information system that efficiently organizes and manages large-sized models in thematic layers was proposed leveraging GPUs parallel programming (López et al., 2020).

Our approach exploits Vulkan's multi-threading capabilities to visualize large 3D models of OBJ format with UV mapped texture coordinates which remains briefly addressed be previous research. It aims at being reliably deployed across the operating systems, architectures and hardware that are compatible with its integrated components and support its minimum computational requirements. For this purpose, techniques for resources allocation, scheduling and adaptation to low-power mobile graphics have been developed and tested.

3. METHODOLOGY

3.1 Rendering pipeline

The execution order of the developed graphics application is presented as follows: The physical and logical devices, the queue families needed to access the inner operations of the API as well as the window surface where the visualization occurs are described on initialization. Then, the image views and the corresponding framebuffers are pre-defined in order to be instantly selected at draw time. The most important part is the graphics pipeline, in which the stages of shaders creation are declared explicitly. It comprises a series of steps required to render objects to the screen where the output of one stage is fed to the input of the next one. The process of transforming the primitives of the 3D mesh to pixels is accelerated on the GPUs which implement functional parallelism to distribute the rendering workload among thousands of computational units. After the description of vertices and indices, clipping and transformations are applied to map the scene to the window viewport. All 3D mesh primitives are converted to fragments and texture coordinates are interpolated from the relative coordinated of the vertices. Depth and stencil tests operate to the rendered pixels of each fragment for post-processing. Finally, the push constants that correspond to a type of small and fast-access uniform buffer memory, send uniform data to the shader. Their functionality serves the need of changing dynamically properties during the drawing phase and in particular, the position and scale of the 3D model when user interacts with it. The state of the pipeline's operations is configured by specific parameters such as the data transfer method, the memory pattern and the reference of the render passes of the desired render target. Each operation is submitted to a queue and recorded to command buffers. They enable the execution of the drawing commands and allow resources to be dispatched to GPU's exclusive memory. The operations of loading the data structures, updating their state and selecting the image views from the command buffer to be presented to the viewport continues to iterate until the application is closed and all data structures and handles are destroyed.

3.2 Synchronization and thread management

The aforementioned operations are executed asynchronously and their submission to the queue needs to be set in order to resolve both scheduling and synchronization. If they do not synchronize, the results can change depending on how events happen to occur. The application's synchronization relies on a Vulkan primitive, the semaphore. Its role is twofold; access of shared resources and control of submission order. They synchronize operations solely on the GPU side and it must be ensured that their state is well defined when it gets signalled from the host device. In the proposed methodology, timeline semaphores are integrated to the algorithms of the queue operations. Such a case is the signal that an image view is acquired from the swap chain in order to be rendered. When the rendering has finished and the presentation can happen relative semaphores are signalled directly from the host. In case of multi-threading, where a single semaphore has to be signalled for multiple threads, the execution of operations is prioritized based on the sequence of host and device queues signal operations. Thus, semaphore programming reduces the synchronization complexity by determining a standard communication protocol and improves the overall's application responsiveness.

The modern CPUs are not anymore single-core. The fact that they are equipped with more than one processing units can be exploited in order to distribute the recording of the drawing commands. The multi-threading scenario of the application resolves the concurrent recording of command buffers, which is a time-consuming operation for the processor. To efficiently record multiple command buffers in parallel, memory access and resources usage are managed per frame and per thread. Prerequisite is the usage of a separate command pool for each thread that allocates a specific command buffer. The preparation of command buffers to render the 3D object is implemented through three stages. The drawing commands for the current frame are incorporating in the main thread through the rendering pipeline while the commands to the secondary command buffer are recorded in a worker CPU's thread. When each process is finished, it is reported to the main thread and specifically, to the primary command buffer. The last operation ends the render pass for current command buffer, reports to the window surface that the frame is ready and the rendering state is updating.

3.3 Tile-based rendering for mobile GPU

The multi-threaded submission in mobile devices is also implemented by command buffers. The optimal approach is the recording of drawing commands in secondary command buffers so as the submission is done to the same render pass. The render passes adapt to the mobile GPU's tiled-rendering that describes the beginning and end of rendering to a framebuffer. Leveraging the tile-local memory, the proposed methodology uses multi-pass render passes for faster tile cache memory on mobile devices. Usually, each pixel is rendered in a subpass and it accesses the results of the previous subpass at the same pixel location. However, in the application, some render passes are merged on the same chip memory like texture mapping and pixels correspondences.

4. IMPLEMENTATION

4.1 Case Study

The input wavefront (OBJ) 3D mesh is part of the geometric documentation of the Archaeological/Holy Site of Meteora, a UNESCO Cultural Heritage site in Greece. The represented landmark is the rock of St. Modestos – Modi on top of which ruins of an old monastery exist. Data were collected using image-based photogrammetric techniques and they were processed with computer vision algorithms, constituting a high-resolution geomatics product. The vertical and oblique aerial images as well as the terrestrial images were oriented through the Structure from Motion (SfM) algorithm and then, a mesh was generated by the sparse point cloud with the Mult-View Stereo (MVS) technique. The process of 3D modelling was undertaken by Agisoft Metashape and Geomagic software. The final 3D model has 4 million vertices, a size of 938 MB, material definitions and a corresponding texture image.

4.2 Multi-threading programming

The prototype integrates third-party libraries like GLFW and Open Asset Import library (assimp). GLFW is an open-source C library, essential for creating a Vulkan surface on initialization and receiving events from windows. Assimp loads, parses and stores 3D model formats in the program-specific format. Once objects are loaded, they are placed in the data structures *m_Mesh* and *m_TextureImage* accordingly, that are handed to Vulkan. Multi-threading enables multiple threads in memory at a given time and switches amongst them in order to provide a pseudo parallelism, assuming that all the threads are executing at the same time. Application's multi-threading technique parallelizes rendering across four CPU threads and two levels of command buffers. The primary command buffer records the work to be conducted by the GPU with big state changes while the secondary command buffer aims solely at building and dispatching draw calls within a render pass. The latest drawing phase starts with uniform buffer generation for each swapchain image, render pass creation and binding of the graphics pipeline object and its resources, including vertex buffers and descriptor sets. In case of window resizing, swap chain's buffer size and number of buffers change accordingly. Figure 1 illustrates the function that records all of these commands on the secondary command buffer. The drawing objects, namely the 3D model and the texture, are submitted in the primary buffer by reusing and executing repeatedly the created drawing calls across the four CPU threads. For texture sampling in the shader, a shader resource binding is created while 3D model's material lighting is already baked into the texture, so there is no need for specular or diffuse lighting utilization. While it is important to record multiple command buffers on multiple threads for efficiency, synchronization primitives are needed to order their execution. Timeline semaphores insert dependencies between queue operations to ensure that display will only take place after the command buffer has finished processing. The draw command waits on a semaphore that determines when rendering can start and signals another semaphore that triggers the display of the finished frame. The 3D viewer's window with the final 3D scene in Visual Studio's IDE (Windows 10 OS) is presented in Figure 2a. The 3D model is visualized with high-fidelity and only a minor visual quality loss in texture mapping is observed.

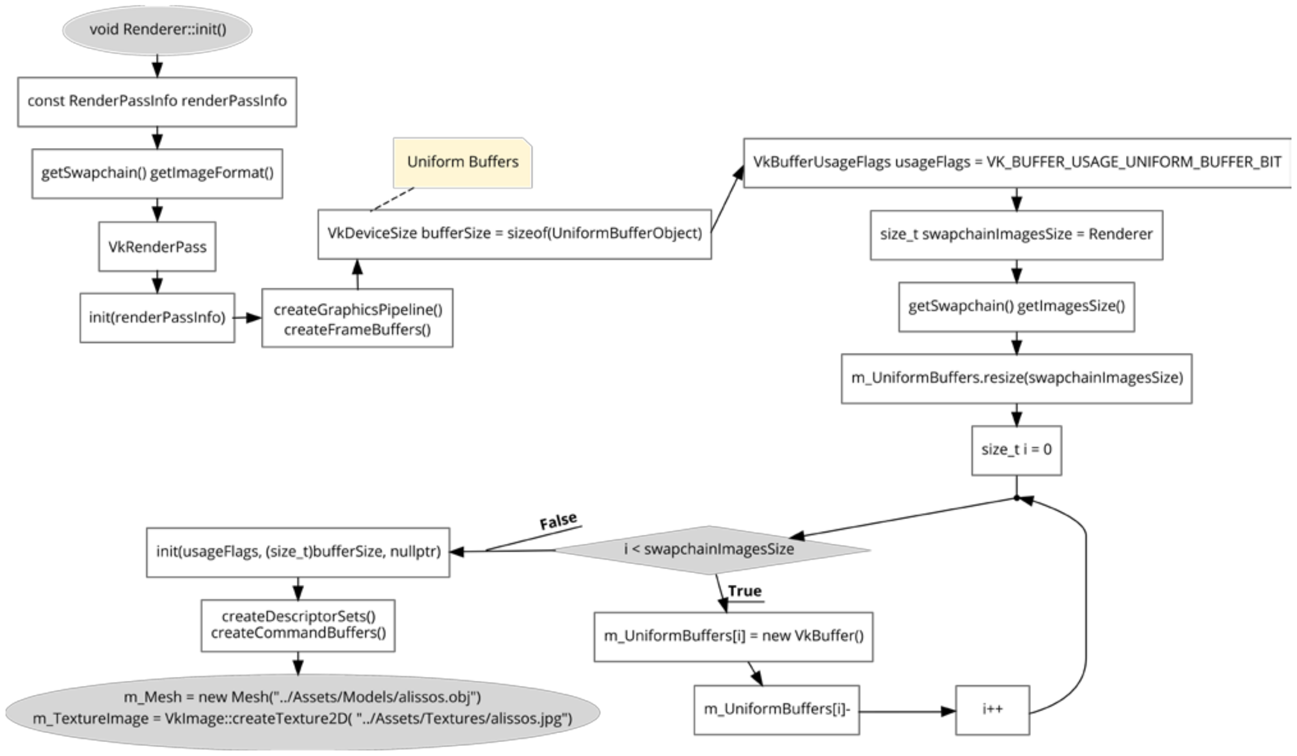
void Renderer::init()
const RenderPassInfo renderPassInfo
getSwapchain() getImageFormat()
VkRenderPass
init(renderPassInfo)
createGraphicsPipeline()
createFrameBuffers()
Uniform Buffers
VkDeviceSize bufferSize = sizeof(UniformBufferObject)
VkBufferUsageFlags usageFlags = VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT
size_t swapchainImagesSize = Renderer
getSwapchain() getImagesSize()
m_UniformBuffers.resize(swapchainImagesSize)
size_t i = 0
i < swapchainImagesSize
False
True
init(usageFlags, (size_t)bufferSize, nullptr)
createDescriptorSets()
createCommandBuffers()
m_Mesh = new Mesh("../Assets/Models/alissos.obj")
m_TextureImage = VkImage::createTexture2D("../Assets/Textures/alissos.jpg")
m_UniformBuffers[i] = new VkBuffer()
m_UniformBuffers[i]-
i++

Figure 1. Initialization of render pass commands in the secondary command buffer where the drawing objects (m_Mesh and m_TextureImage) are defined.

4.3 Multi-platform support

Using a single codebase, the prototype is not tied to a single operating system, GPU vendor or architecture. To illustrate portability, the 3D viewer is also built for MacOS and Android. Vulkan is not supported by Apple devices but the MoltenVK runtime library is used to convert SPIR-V shader code to Metal Shading Language (MSL) and run the developed application across macOS platforms. The library maps Vulkan's functionality to Apple's Metal graphics framework and the 3D viewer runs from Xcode IDE (Figure 2b). Finally, most Android devices use tile-based rendering. The prototype adapts the multi-threading model to this logic by setting the load/store operations of the render passes explicitly and parallelizing recording of draws in a pass. On most mobile GPU architectures, beginning and ending a render is an expensive operation. The configuration is done in the low-latency memory on the GPU to reduce this computational cost. The native source code is wrapped into a library and the rest of the development is handled by Android Studio IDE and JNI framework (Figure 2c). In the 3D mobile viewer, a multitouch gesture interface is created to let users inspect every part of the 3D model.

5. EVALUATION

A performance evaluation is conducted by testing the three different implementations of the same codebase on the following operating systems: Windows 10, MacOS 10.15.14 and Android 9. It aims at examining the efficacy of the developed rendering techniques and synchronization strategies rather than providing a comparative analysis between the various platforms. Incorrect usage of multithreading may result in high CPU usages or increased CPU cycles which could drastically reduce application's performance. Benchmarking is conducted by the diagnostic tools of each platform's IDE. The tests run for 170 seconds and the recorded performance metrics are reported as average frames per second (FPS) and diagrams with CPU and GPU activity. One of the key metrics, exported from the diagrams, is the total CPU usage of the 4 threads. The average number of FPS is calculated once the visualization is completed. The multi-threaded rendering times are affected by the hardware's host system as well as the size of OBJ and texture files. The experimental setup and the test results are presented in Table 1. Additional benchmarking concerns the workload division across multiple CPU threads and examines the impact of the Windows's implementation in case of a specific number is activated. The time spend to render the entire scene is measured and presented in Table 2.

	Hardware specifications		**Test Results**	
	GPU Type/Memory	CPU	FPS	Total CPU usage
Windows 10	NVIDIA GeForce RTX 2070, 8 GB GDDR6	3.60 GHz, 8-Core	145	22,57
MacOS 10.15.4	AMD Radeon Pro 555X, 4 GB GDDR5	2.2 GHz, 6-Core	117	29,88
Android 9	Qualcomm Adreno 610	2.0 GHz Kryo 260, 8- Core	52	33,42

Table 1. Experimental setup and part of the performance results for each operating system and device's hardware

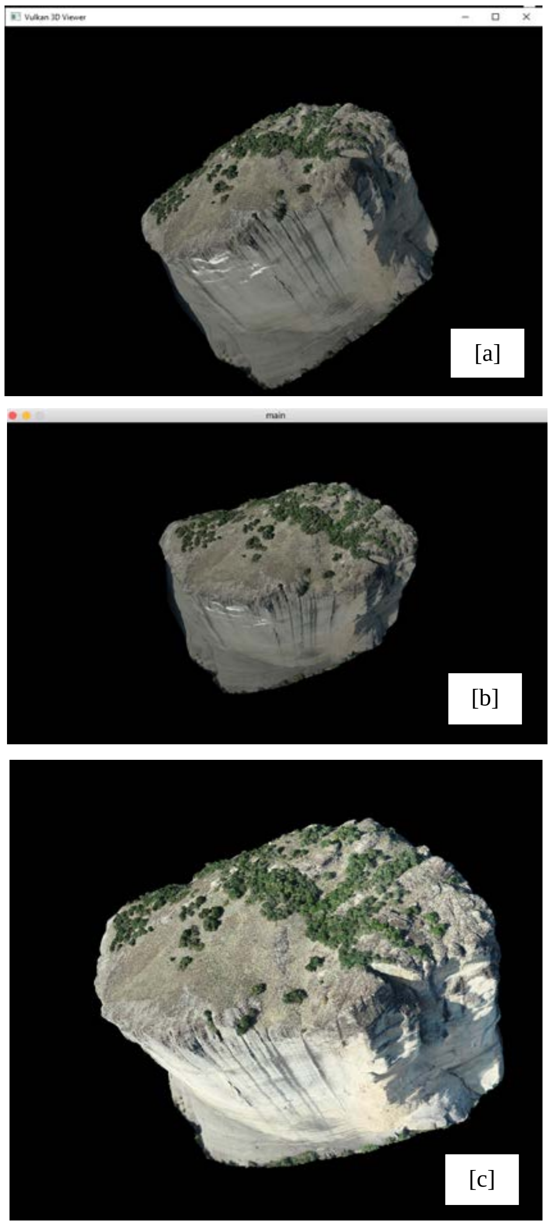

Figure 2. Visual output of the Vulkan-based 3D rendering on (a) Windows 10, (b) MacOS 10.15.14 and (c) Android 9.

Number of Threads	Time (ms)
Single-threaded	454.07
Two	233.85
Three	173.52
Fours	125,22

Table 2. Number of threads enabled and total rendering time until the 3D mesh is visualized.

The performance evaluation varies greatly depending on which hardware is tested and what drawing objects are used. As it is expected, running the prototype on a more powerful graphics card results in a higher number of FPS. Even if the MacOS and Android CPUs are less powerful, frame rates are high. This indicates that utilizing multiple threads for command buffer generation significantly increases the performance of a program that is CPU limited. The total CPU usage is low while the GPU undertakes the biggest part of the graphics workflow. Performance improves approximately linearly with the number of cores in the system. Results of Table 2 indicate that the workload is divided well across the four threads without incurring much additional processing costs. In case the user's CPU is equipped with four cores, the performance benefit is significant.

6. CONCLUSIONS

The developed 3D viewer demonstrates great portability to a multitude of devices and platforms and high degree of performance stability. It adapts to the implicit tile-based rendering of mobile GPUs, scaling from low-power mobile devices to high-end workstations. The compatibility with non-dedicated hardware, the ability to handle large datasets and the visual quality will compensate for the lack of high-end technological resources of institutes, communities and research groups. Prioritizing performance over ease of use, the application can serve visualization cases with high-computational demands, like cultural heritage monuments or sites, 3D cadastral and urban planning datasets, LiDAR data, 3D scanning products, etc. Rendering, shading, lighting or even memory and resources allocation can be defined explicitly for customized appearance and adaptation to the researcher's specific needs. The 3D viewer currently supports OBJ files loading with a single texture image but with the integration of assimp library multiple 3D data formats will be supported in the future. The GPU acceleration and multi-threading technique are fast-growing areas that generate a lot of interest from researchers and scientists that develop computationally intensive applications. Therefore, the proposed synchronization and workload distribution techniques be used as building blocks for any visual application on Vulkan API. Future work includes the integration of the ray-tracing option for photorealistic textures and advanced postprocessing effects. The application will be able to alternate between rasterization and ray tracing rendering depending on the task and current graphics card capabilities.

ACKNOWLEDGEMENTS

This research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EΔK-02859). The authors would like to thank Ordnance Survey GB (https://www.ordnancesurvey.co.uk) and 1Spatial (https://1spatial.com/) for sponsoring the publication of this paper.

REFERENCES

Blackert, A., 2016. Evaluation of multi-threading in Vulkan. https://hgpu.org/?p=16886. (17 July 2020).

Cho, S., Hong, J., Choi, J., Han, H., 2019. Multithreaded double queuing for balanced CPU-GPU memory copying. *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 1444–1450. doi.org/10.1145/3297280.3297426

Feinbube, F., Troger, P., Polze, A., 2011. Joint Forces: From Multithreaded Programming to GPU Computing. *IEEE Software*, *28*(1), 51–57. doi.org/10.1109/MS.2010.134

Gambhir, M., Panda, S., Basha, S. J., 2018. Vulkan rendering framework for mobile multimedia. *SIGGRAPH Asia 2018 Posters*, 1–2. doi.org/10.1145/3283289.3283336

Grigg, R. J., & Hexel, R., 2017. Communication versus computation: A survey of cloud gaming approaches. *18th International Conference on Intelligent Games and Simulation*, 10-25

Gunadi, S. I., Yugopuspito, P., 2018. Real-Time GPU-based SPH Fluid Simulation Using Vulkan and OpenGL Compute Shaders. *2018 4th International Conference on Science and Technology (ICST)*, 1–6. doi.org/10.1109/ICSTC.2018.8528699

Heitzler, M., Lam, J. C., Hackl, J., Adey, B. T., Hurni, L., 2017. GPU-Accelerated Rendering Methods to Visually Analyze Large-Scale Disaster Simulation Data. *Journal of Geovisualization and Spatial Analysis*, 1(1–2), 3. doi.org/10.1007/s41651-017-0004-4

Lavrič, P., Bohak, C., Marolt, M., 2018. Vulkan Abstraction Layer for Large Data Remote Rendering System. *De Paolis L., Bourdot P. (eds) Augmented Reality, Virtual Reality, and Computer Graphics. AVR 2018. Lecture Notes in Computer Science, vol 10850.* Springer International Publishing. doi.org/10.1007/978-3-319-95270-3_40

López, L., Torres, J. C., Arroyo, G., Cano, P., Martín, D., 2020. An efficient GPU approach for designing 3D cultural heritage information systems. *Journal of Cultural Heritage*, 41, 142–151. doi.org/10.1016/j.culher.2019.05.003

Mittal, S., 2014. A SURVEY OF TECHNIQUES FOR MANAGING AND LEVERAGING CACHES IN GPUs. *Journal of Circuits, Systems and Computers*, 23(08), 1430002. doi.org/10.1142/S0218126614300025

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., Phillips, J. C., 2008. GPU Computing. *Proceedings of the IEEE*, 96(5), 879–899. doi.org/10.1109/JPROC.2008.917757

Pezhgorski, V., Lazarova, M., 2017. Real Time GPU Accelerated Radar Scan Conversion and Visualization. *Proceedings of the 18th International Conference on Computer Systems and Technologies*, 249–256. doi.org/10.1145/3134302.3134339

Piao, J.-C., Cho, C.-W., Kim, C.-G., Burgstaller, B., Kim, S.-D., 2014. An Adaptive LOD Setting Methodology with OpenGL ES Library on Mobile Devices. *2014 International Conference on IT Convergence and Security (ICITCS)*, 1–4. doi.org/10.1109/ICITCS.2014.7021727

Redlarski, J., Trzosowski, R., Kowalski, M., Kowalski, B., Lebiedź, J., 2018. Stereoscopy in Graphics APIs for CAVE Applications. 2018 *Federated Conference on Computer Science and Information Systems (FedCSIS)*, 893–896. doi.org/10.15439/2018F223

Rogers, T. G., O'Connor, M., Aamodt, T. M., 2014. Learning your limit: Managing massively multithreaded caches through scheduling. Communications of the ACM, 57(12), 91–98. doi.org/10.1145/2682583

Thoman, P., Wippler, M., Hranitzky, R., Fahringer, T., 2020. RTX-RSim: Accelerated Vulkan Room Response Simulation for Time-of-Flight Imaging. *Proceedings of the International Workshop on OpenCL*, 1–11. doi.org/10.1145/3388333.3388662

Wu, J., Deng, L., Paul, A., 2015. 3D Terrain Real-time Rendering Method Based on CUDA-OpenGL Interoperability. *IETE Technical Review*, *32*(6), 471–478. doi.org/10.1080/02564602.2015.1040473

Zhang, A., Chen, K., Johan, H., Erdt, M., 2018. High performance city rendering in Vulkan. *SIGGRAPH Asia 2018 Posters*, 1–2. doi.org/10.1145/3283289.3283342